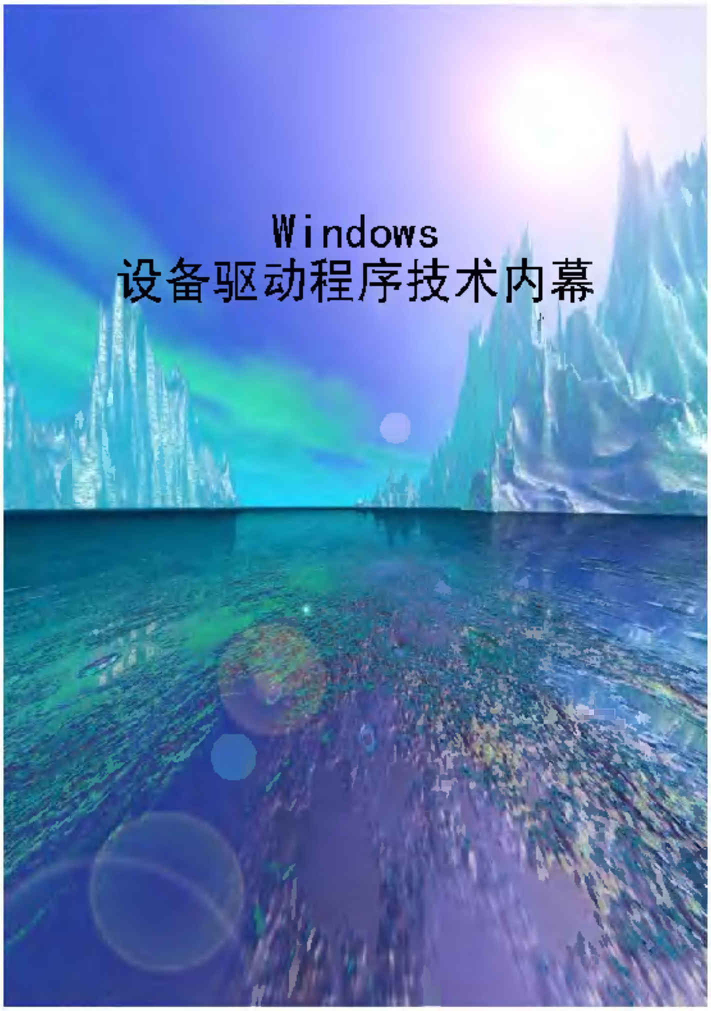
Windows
设备驱动程序技术内幕

Windows 设备驱动程序技术内幕

孙守阁 徐 勇 编

清 华 大 学 出 版 社

（京）新登字 158 号

内 容 摘 要

Windows 设备驱动程序技术内幕一书，详细讲解了设备驱动程序的原理及实现方法，并例举了大量的程序实例，便于读者学习和掌握。

本书分为三个部分。第一部分主要讲怎样用 DOS 的方法写设备驱动程序，该部分的驱动程序被封装为 DLL 形式。第二部分讲怎样写 Windows 的虚拟设备驱动程序（VxD）。第三部分介绍 Windows 系统下标准模式设备驱动程序的编写方法。

本书是一本技术性较强的工具书，它主要针对有一定计算机基础的程序开发者和硬件设计者，同时也适应需要进一步了解计算机应用的读者。

版权所有，翻印必究。

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

书　　名：Windows 设备驱动程序技术内幕

作　　者：孙守阁　徐　勇

出 版 者：清华大学出版社(北京清华大学学研楼，邮编 100084)

http://www.tup.tsinghua.edu.cn

责任编辑：唐　玲

印 刷 者：北京市通州区大中印刷厂

发 行 者：新华书店总店北京发行所

开　　本：787×1092 1/16　**印 张**：14　**字 数**：331 千字

版　　次：2000 年 5 月第 1 版　2000 年 6 月第 2 次印刷

书　　号：ISBN 7-302-00900-7/TP・329

印　　数：5001～10000

定　　价：22.00 元

前　言

现在市面上关于 Windows 编程的书很多，但是关于 Windows 设备驱动程序编程的书很少，而关于虚拟设备驱动程序的编程就更少了。在工作中往往需要对标准的硬件设备进行操作，有时还需要使用设备驱动程序对物理设备进行操作，这就需要编程人员掌握这些方面的知识，能够设计出自己需要的设备驱动程序。

本书主要介绍了 3 种设备驱动程序的编写方法。全书分为 3 部分，每一部分先介绍一些基础知识，再进行详细内容的讲述。各部分的内容都是循序渐进的，便于学习和掌握。

第一部分主要讲述怎样用 DOS 的方法编写设备驱动程序。该部分的驱动程序被封装为 DLL 形式，可以被 Windows 应用程序访问，但不能被 DOS 应用程序访问。

第二部分讲述怎样编写 Windows 的虚拟设备驱动程序（VxD）。这些虚拟设备驱动程序作为被托管的 Windows 核心成员运行，可以“到任何地方做任何事”，最终可以知道 Windows 是怎样实现虚拟机的。它们的结构十分直观，便于了解和掌握。当运行环境变得复杂时，VxD 本身和基于 DLL 的驱动程序同样简单。

第三部分介绍 Windows 系统下标准模式设备驱动程序的编写。这一部分讲述了怎样编写一个高质量的驱动程序，Windows 程序怎样与 DOS 的 TSR 进行通信，在 Windows 中怎样用 DMA 进行工作，以及其他一些高级应用。

本书是一本技术性较强的工具书，它主要适用于有一定计算机基础的程序开发者和硬件设计者，同时也适应于需要进一步了解计算机应用的读者。

编　者

1999 年 12 月

目　　录

第 1 章　概述 1
1.1　本书的读者对象 1
1.1.1　哪些人应该读这本书 2
1.1.2　哪些人不能读这本书 2
1.2　应用平台 2
1.3　本书的结构 3
1.4　学习的方法 3
1.5　需要的工具 3

第一部分　驱动程序基础

第 2 章　基础知识 7
2.1　驱动程序的定义 7
2.2　特权封装 7
2.3　非特权封装 7
2.4　DLL 与静态库的比较 8
2.5　为什么要把驱动程序封装为 DLL 8
2.6　应用程序与 DLL 比较 9
2.6.1　DLL 和栈段 9
2.6.2　DLL 和数据段 10
2.6.3　DLL 和动态分配内存的主权 10
2.6.4　DLL 的初始化和终止 11
2.7　DLL 的函数要求 11
2.8　框架驱动程序 12
2.9　建立框架驱动程序 14
2.10　DLL 需要应用程序 16
2.11　驱动程序的调试工具 17
2.12　总结 17
第 3 章　与硬件相连接 18
3.1　端口映射和内存映射的比较 18
3.1.1　访问端口映射硬件 18
3.1.2　访问内存映射硬件 18

3.1.3 两步骤地址转换过程 19
3.1.4 对 1MB 以下的设备映射使用预定义的选择器 21
3.2 端口映射例子程序 21
3.2.1 例子程序 3.1 22
3.2.2 例子程序 3.2 22
3.2.3 例子程序 3.3 23
3.2.4 例子程序 3.4 28
3.2.5 例子程序 3.5 28
3.3 驱动程序设计概述 29
3.4 驱动程序例子 30
3.5 内存映射的方案 32
3.6 先进内存的问题 32
3.7 设备映射大于 1MB 需要 DPMI 服务 34
3.8 总结 36
第 4 章 中断方式设备驱动程序 37
4.1 Windows 虚拟内存 37
4.2 实现方法 38
4.3 什么是可以丢弃的 38
4.4 为什么整理内存 39
4.5 安全中断驱动程序内存需求 41
4.5.1 固定的原因 41
4.5.2 不可丢弃的原因 41
4.5.3 页面锁定的原因 42
4.6 中断安全的代码和数据 42
4.6.1 FIXED 何时是真正固定的 43
4.6.2 动态分配中断安全的缓冲区 43

第二部分　虚拟驱动程序简介

第 5 章 Windows 的虚拟世界 47
5.1 虚拟机是什么 47
5.2 Windows 的执行环境 48
5.2.1 管理程序 48
5.2.2 Windows 应用程序 48
5.2.3 DOS 应用程序 48
5.3 Windows 地址空间 49
5.3.1 线性地址空间 49

5.3.2 线性地址空间和物理地址空间 51
5.3.3 逻辑、线性和物理地址空间 52
5.4 这意味着什么 52
5.5 实现虚拟机 53
5.5.1 捕获 I/O 操作 53
5.5.2 捕获内存操作 54
5.5.3 捕获中断和异常事件 54
5.6 V86 模式简介 55
5.7 总结 56

第 6 章 虚拟驱动程序简介 57
6.1 VxD 的基本结构 57
6.2 设备描述器部件 58
6.3 支持数据结构 59
6.4 事件的标志信息 61
6.5 保护模式初始化 62
6.6 创建 VxD 的工具 63
6.7 总结 64

第 7 章 设备选优 VxD 65
7.1 VMM 和处理器的异常事件 65
7.2 端口所有权 66
7.2.1 PORTTRAP（端口访问）程序例子 66
7.2.2 仿真多字节 I/O 68
7.2.3 安装和使用 PORTTRAP 69
7.3 例子 PAGETRAP 70
7.3.1 初始化例程 70
7.3.2 错误处理器例程 71
7.3.3 结束处理器 72
7.3.4 测试 PAGETRAP 73
7.4 总结 73

第 8 章 服务硬件中断的 VxD 83
8.1 中断和 VMM 83
8.1.1 中断反射的过程 84
8.1.2 哪个 VM 获得中断 85
8.1.3 扩展模式中的中断执行时间 85
8.2 VxD 通信 86
8.2.1 传递参数 86

8.2.2 从应用程序调用 VxD 的 API 87
8.2.3 硬件中断处理 VxD 87
8.2.4 API 调用服务 89
8.3 硬件中断反射 VxD 90
8.3.1 回调 Mast 90
8.3.2 回调 Hw_Int 91
8.3.3 回调 EOI 91
8.3.4 回调 Virt_Int 和 IRET 92
8.4 总结 92

第三部分　特殊技术

第 9 章　第 0 层处理器缩短响应时间 111
9.1 替换 VxD 111
9.1.1 安装 112
9.1.2 初始化 BIMODAL_INT_STRUC 112
9.2 注册处理器 114
9.2.1 VPICD 初始化字段 116
9.2.2 处理中断 117
9.3 编写处理器程序 117
9.3.1 为什么没有堆栈变量 118
9.3.2 一种更糟的情况 119
9.4 总结 120
第 10 章　双模态驱动程序 139
10.1 新的双模态驱动程序综述 139
10.1.1 新的双模态驱动程序代码 140
10.1.2 回调细节 144
10.2 总结 145
第 11 章　编写使用 DMA 的驱动程序 166
11.1 DMA 缓冲区的要求 166
11.1.1 物理上毗邻 166
11.1.2 固定的和页面锁定 166
11.1.3 定位在 64KB 的边界上 167
11.1.4 怎样分配 DMA 缓冲区 168
11.2 Windows 下 DMA 的 DOS 应用程序 168
11.3 DMA 的 Windows 应用程序可使用这个知识 169
11.3.1 使用 VDS 缓冲区更好 169

11.3.2 最佳的解决方案——助手 VxD 171
11.4 DMABUF 的 API 172
11.5 使用助手 VxD——USEAPI.C 173
11.6 总结 173

第 12 章 使用实模式服务 181
12.1 关于 DOS 设备驱动程序 182
12.1.1 对 IOCTL 的专门处理 183
12.1.2 介绍 DOS TSR 184
12.1.3 通过缓冲区传递数据 185
12.1.4 TSR 调用 Windows 应用程序 189
12.2 增强模式中的回调 189
12.3 编程细节 190
12.4 总结 193

第 13 章 编写标准模式的驱动程序 194
13.1 基本概念 194
13.1.1 硬件的端口映射 194
13.1.2 内存映射的硬件 194
13.1.3 标准模式中的选择器 195
13.2 标准模式中的中断 196
13.3 从保护模式和实模式中获取中断 197
13.4 分离实模式控制器和保护模式控制器 198
13.5 总结 200

第 14 章 定时器和软件中断 206
14.1 定时器 206
14.1.1 使用 SetTimer（） 206
14.1.2 调用 INT 1CH 和 INT 8H 206
14.1.3 不要依赖于每秒 18.2 次 207
14.1.4 使用 timeSetEvent() 207
14.1.5 使用 VxD 207
14.1.6 不要使用 GetTickCount() 207
14.1.7 使用 BIOS 的周期计数或者是 timeGetTime( ) 208
14.2 软件中断 208
14.2.1 连接到软件中断 208
14.2.1 用 VxD 捕获软件中断 209
14.3 编程细节 210
14.4 总结 210

第 1 章　概　　述

现在有许多关于 Windows 编程的书，但很少有关于 Windows 设备驱动程序编程的书，而关于虚拟设备驱动程序的编程就更少了。如果在工作中需要对标准的硬件设备进行操作，而且 Windows 使用设备驱动程序来对物理设备进行操作，就需要编程人员掌握这方面的知识，并对这方面的具体应用有所了解。以前的解决办法是设备生产商将他们的驱动程序开发人员送到 Microsoft 学校，进行专门地培训，从而完成所需驱动程序的开发工作。而一般的用户需要使用相应的 Windows 应用程序接口（API）来与驱动程序通信。

但在实际应用中，并非所有的设备都是标准设备。特别是当 Windows 系统的使用越来越广泛的时候，许多领域的专业编程人员发现需要自己来开发驱动程序，以满足对 Windows 应用程序开发的需要。

此外，有些不与非标准设备直接通信的开发者，也需要知道怎样编写一种特殊的设备驱动程序，如利用虚拟设备驱动程序（VxD）来实现自己设计的要求。现在 VxD 已被广泛应用，在 Windows 环境中，VxD 是开发者扩展操作系统的工具——“到任何地方做任何事”。如果你想编写功能强大的调试代码，想改变 Windows 初始化的方法，在 DOS 应用程序和 Windows 应用程序之间通信，编写高性能的驱动程序，你就需要知道怎样创建 VxD，这也是编写本书的目的。

1.1　本书的读者对象

如果你要创建支持非标准设备的驱动程序，你就需要读这本书。标准设备是指计算机的标准配置，如显式器、键盘、鼠标、串口或打印机等的设备；非标准设备是指除标准设备之外的一切设备。这些设备是各个用户根据自己的需要设计的，它与计算机之间没有固定的连接方式，在 Windows 系统中也没有指定这类设备的软件接口，所以它与计算机的连接可以有几种方式选择。本书将逐一介绍这些可供选择的方法，从最简单的动态连接库（DLL）到复杂的双模态中断处理器。本书还将介绍怎样使 Windows 应用程序、DOS 的内存驻留程序（TSR）与 DOS 设备驱动程序进行通信等内容。

本书的读者对象是设备接口的程序开发者。如果你不明白其他程序开发者怎样使他们的程序在 Windows 系统工作起来，你就需要读这本书。如果你想明白一个调试器怎样捕获指针错误；还不明白怎样访问硬件，Windows 是怎样将 DOS 应用程序的结果输出到一个屏幕窗口，怎样编写 Screen Grabbers；想了解有的设备驱动程序在 Windows 启动时检测遗漏的硬件，而有的驱动程序只能在应用程序开始的时候检测遗漏的硬件的原因等，并想解决它，你就需要读这本书。

所有这些问题的答案是虚拟设备驱动程序 VxD（以下简称 VxD），它允许你将新的

功能直接加入到操作系统中。本书将告诉你怎样写 VxD，同时还告诉你如何用 VxD，而不被 Windows 的琐事和 API 访问所干扰，还会告诉你什么对于 VxD 是重要的，什么是你不需要知道的。

1.1.1 哪些人应该读这本书

如果你曾经在 DOS 或其他操作系统下编写过设备驱动程序或者设备接口程序，你适合读这本书。为了从本书中得到更多的东西，你需要对 C 语言和汇编语言非常熟悉，而且还要知道怎样使用 INT 21H 调用 DOS 的应用，熟悉掌握“段”怎样被 DOS 编译器和汇编器使用，你才能很好地读完这本书。

读这本书，不必是一个 Windows 应用程序编程人员。实际上，本书中的程序类似于传统的 DOS 程序。由此可见，预先具备 DOS 编程的经验是有用的，但不是必须的。

1.1.2 哪些人不能读这本书

本书对初学者不适用。要学习这本书，必须熟悉汇编语言和 C 语言，而且对操作系统和计算机结构有一定的了解和经验。例如本书解释了怎样在 Windows 应用程序中正确分配和寻址直接内存访问（DMA），但没有解释什么是 DMA；介绍了怎样在 Windows 下实现一个临界区域，但没有解释什么是临界区域，以及为什么需要它。

本书不是在讲述怎样编写标准的 Windows 设备驱动程序，而是讲述怎样编写非标准的设备驱动程序或者自己所需的特殊设备驱动程序。如果你只想编写一个打印机驱动程序或者通信接口驱动程序，你就应该读其他的书。虽然本书的内容可以直接用于标准的驱动程序，但是专门关于标准设备驱动程序的书会更有用。当然，你也可以先读本书，再读关于标准驱动程序的书。从这个意义上讲，本书是关于所有 Windows 驱动程序的，因为标准的驱动程序只是非标准驱动程序中的一种特殊例子。本书的内容不包括写磁盘或 CD－ROM 这样的数据块驱动程序。

1.2 应用平台

本书的应用在 Windows 平台上。如果没有另外的说明，它适用于 Windows 3X 和 Windows 9X 平台。本书的前两部分通常忽略 Windows 的标准模式应用。由于 PC 286/386 的机器在市场上变得越来越少，开发者主要选择的是提供增强模式的支持。由于硬件趋势以及非标准设备趋于高级化，使得标准模式对于设备驱动程序的编写者变得越来越不重要，而对于非标准模式的设备驱动程序的编写却显得越来越重要。

如果读者学会了用增强模式（非标准模式）来对硬件进行操作，就会更容易理解标准模式的问题。因为标准模式是非标准模式的一种特殊情况。正因为这个原因，本书直到第三部分才讨论标准模式的问题。

虽然 Windows 9X 在结构上与 Windows 3X 有所不同，但是本书中的大部分内容适用于 Windows 9X。虽然许多标准驱动程序的 API 已经改变，而且数据块设备也适用 VxD 设备驱动程序，但是许多非标准的 Windows 3X 驱动程序可以不需要任何改变，就能在 Windows9X 下运行。

1.3 本书的结构

本书分为三个部分，每一部分都介绍一些后面所需的基础知识，再进行详细地讲述。而且各部分的内容都是循序渐进的，便于学习和掌握。

第一部分主要讲的是怎样用 DOS 的方法编写设备驱动程序。该部分的驱动程序被封装为 DLL 形式，它可以被 Windows 应用程序访问，但不能被 DOS 应用程序访问。

第二部分讲的是怎样编写 Windows 的虚拟设备驱动程序。这些虚拟设备驱动程序作为被托管的 Windows 核心成员运行，可以“到任何地方做任何事”。当然，这些驱动程序需要学习者详细了解和掌握，最终可以知道 Windows 是怎样实现虚拟机，而它们的结构又十分直观，便于了解。当运行环境变得复杂时，VxD 本身和基于 DLL 的驱动程序一样很简单。

第三部分介绍了 Windows 系统下标准模式设备驱动程序的编写。这一部分讲述了怎样编写一个高质量的驱动程序，Windows 程序怎样与 DOS 的 TSR 进行通信；在 Windows 中怎样用 DMA 进行工作，以及其他一些高级的应用。

1.4 学习的方法

Windows 可能是一个非常复杂的环境。这本书的主要目的是帮助你理解在这个环境中，对于不同类型的驱动程序，它的关键是什么。每一章介绍一个新的驱动程序或者与之相关的应用程序，而且每一章只介绍能够帮助、理解新例子的有关资料。尽量使每个例子的驱动程序简单，以便于关键的特性一看就明白。每一章都按照循序渐进排列，每个例子都在前一章的基础上建立。所以读者在学习这本书时需要从头到尾进行，只有这样才能很好地理解和掌握本书的内容。有条件的读者在看书的同时，可以结合书上的例子，进行一些实际的操作，这样收获会更大。

本书中的大部分例子程序，都是用 Microsoft C v8.0 编写的，不过在有一些地方使用了汇编语言 Microsoft Assembly（MASM）v6.0 编写。

1.5 需要的工具

对于编写基于 DLL 的驱动程序，需要准备一个可以生成 Windows 应用程序的编译器，

如果编译器可以嵌入汇编语言则更好。对于编写 VxD 程序，则需要一个专门的汇编器和连接器。它是 Windows Device Developer Kit（DDK）的一部分，只能通过定购 Microsoft Developer Network 得到。本书提供的测试驱动程序还使用了一个工具（DBWIN.EXE），它也是 Windows Device Developer Kit（DDK）的一部分，能够对所编写的程序进行方便地测试和运行。

第一部分

驱动程序基础

第2章 基础知识

本章的主要内容是通过建立一个标准的开发环境，为本书中的其他部分作准备。在本章中解释什么是驱动程序以及怎样去建立它，同时介绍一个在本书中使用的测试应用程序，并且介绍其他将要用到的调试程序。本章中介绍的框架驱动程序是第一部分中所有驱动程序的基础。

2.1 驱动程序的定义

从广义上讲，驱动程序是指一系列控制硬件设备的函数。一个驱动程序分类的方法是看这些函数是怎样去封装的。在 DOS 系统中，一个驱动程序可能是一个连接到应用程序.EXE 中的一个模块，或者是软件中与应用程序完全分开的另一部分（一个 TSR）。封装 Windows 驱动程序的最好方法是制作一个 DLL。DLL 是上述两种封装方式的交叉，它虽然连接到应用程序上，但它实际上又不是应用程序的某一部分。

2.2 特权封装

有一些操作系统如 OS/2 和 UNIX，在应用中禁止应用程序直接支持硬件。在这种环境下，只有被称为设备驱动程序的代码才能被允许与硬件相连。控制硬件的应用程序必须使用这些驱动程序所提供的服务程序。严格地讲，应当尽量地为这些构造的、特权的设备控制模块保留设备驱动程序的内容。Windows 系统下的 VxD 就属于这种情况。

2.3 非特权封装

虽然可安装的设备驱动程序已被 DOS 2.0 及其以上的版本所支持，但真正用于 DOS 环境下的驱动程序几乎只用于大容量存储器（如硬盘、软盘和 CD-ROM）以及网络适配器。对于那些非标准接口的硬件，DOS 应用程序一般使用 TSR。这些 TSR 有时被归类为驱动程序，其中的一些惯例也用于 Windows 中。

在 Windows 环境中，驱动程序通常是指与一个 Windows 系统认为是标准设备相连的 DLL。Windows 为这些标准设备类型定义了不同的驱动程序接口。关于这些设备驱动程序更多的信息，可参见 Daniel Norton 的书《编写 Windows 设备驱动程序》（见参考目录）。

Windows 不支持其他类型设备的接口，禁止将支持硬件的 TSR 以及 DOS 设备驱动程序直接放入 Windows 应用程序中。

虽然你可以把设备接口代码直接放入 Windows 应用程序中，但是通常都把这样的支持封装在 DLL 中。

在本书中使用的驱动程序内容是指一些与非标准硬件、TSR 或 DOS 驱动程序相连的 Windows 代码，这些代码可以被封装成一个模块连到应用程序中，比如一个独立的 DLL 或一个特权级的 VxD。这一部分中重点介绍被封装成 DLL 的驱动程序。

2.4　DLL 与静态库的比较

虽然 DLL 可以像一个静态库一样被应用程序连接和执行，但它并不真正是某一单个应用程序的一部分。了解 DLL 与静态库的区别，将有助于理解为什么驱动程序被封装为 DLL 有用并且是一种非常实用的方法。

静态库（如 C 编译器的运行期库）是指一个或多个预编译的函数。静态库被封装为一个单一的部分，连接器从中提取必要的部件。在连接时，连接器搜索程序来查找应用程序之外的函数，并且通过在库文件中搜索这些函数来实现查询。接着库函数就被拷贝到你的应用程序的.EXE 映象中，并且在连接后，应用程序调用库函数与调用它自己的内部函数完全一致。

DLL 也是一组被预编译的函数。当这些函数被封装时，生成两个部分：一个入口库文件（.LIB）和一个 DLL 文件。在 DLL 中包含了实际的代码和数据，入口库文件只包括函数的名字和模块信息。一个应用程序如果要调用 DLL 中的函数，它就要与入口库连接，而不是与 DLL 连接。连接器并不分解应用程序查询到 DLL 中的函数，也就是说，连接器并不拷贝 DLL 函数到应用程序的.EXE 映象中。相反，连接器只把入口库中的函数名和模块作为位置标志符存在.EXE 中。

当应用程序调用时，真正的奇迹就在运行时发生了。加载器也将 DLL 调入内存，给所有的 DLL 函数一个地址。然后加载器返回.EXE，并将连接时留下的位置标识符填充为 DLL 函数的地址。

2.5　为什么要把驱动程序封装为 DLL

DLL 是封装驱动程序的一个最简便方法，因为一个驱动程序经常被多个应用程序使用，经常需要应用程序的不断变换。由于驱动程序位于一个与应用程序分离的文件中，因此它可以在不破坏驱动程序的应用程序的情况下自动更新。即使一个驱动程序同时被多个应用程序使用，在内存中也只有一个 DLL 的拷贝，而作为静态库的驱动程序则意味着每个使用它的应用程序都有属于自己的一个拷贝。

把 Windows 驱动程序封装为 DLL 的最重要原因是驱动程序生成操作系统调用的确定类型，而应用程序却不能。更准确地说，Windows 几乎不支持这些动作，只有当它们被做成 DLL 形式以后，Windows 才能支持。

2.6 应用程序与 DLL 比较

DLL 与应用程序在某些基本方面存在不同，这些不同与 DLL 的编制和建立有关。对用户来说，最明显的区别是 DLL 不能在程序管理器或其他 Windows 外壳中直接调用。它没有“自己的生命”，但是当使用它的应用程序被调入时，它即被加载；而在应用程序结束时，即被卸载。

从开发者的角度来看，DLL 和应用程序有许多不同，主要表现为以下几方面：

1. 应用程序有自己的栈段，DLL 却没有，它使用的是调用程序的栈段。

2. 如果应用程序的多个事例同时运行，所有的事例分享一个应用程序代码段的拷贝。与此类似，如果多个应用程序同时使用一个 DLL，也只有一个 DLL 代码段的拷贝（这是 DLL 比静态库先进的地方之一）。

3. 如果应用程序的多个事例同时运行，每个事例都有自己的应用程序数据段的拷贝。如果多个应用程序同时使用一个 DLL，也只有一个 DLL 数据段拷贝。

4. 被 DLL 动态分配的内存可以属于应用程序，也可以属于 DLL 本身，需要依据分配的方法来决定。

以上特点在 DLL 的编制和建立的整个过程中，都为开发者进行了区别，下面我们详细讨论上述问题。

2.6.1 DLL 和栈段

一个 DLL 没有它自己的栈段，把指针作为参数传递给其他 DLL 函数将会导致一些小麻烦。如下面的例子所示，它们可以很容易产生一些小错误。

假设一个应用程序有以下的函数：

```
void FAR Foo1(void)
{
    int x;
    Foo2(&x);
}
```

DLL 包括下面的函数：

```
void Foo2(int *x)
{
    int y;
    y=*x;
}
```

如果这些代码作为小模型编译，则被 Foo2 调用的表达式&x 是一个 near 指针。这表明当生成的代码把 X 的地址压入堆栈时，它只把变量 x 地址的偏移量压入堆栈；当 Foo2 从

堆栈中获得这个偏移量并且间接查询它时，它将会不正确地接受这些代码，因为这是一个near 指针，偏移量与 DS（DLL 的数据段有关）有关。Foo2 不知道偏移量实际上与 SS 有关，结果是表达式*x 访问不正确的地址，Y 得到不正确的值。

为什么 DLL 存在这个问题而一般的应用程序没有呢？因为当一个普通应用程序运行时，DS 的值与 SS 的一样，对于 DLL 则 DS!=SS。这不是说 DLL 不能把本地变量的地址作为一个参数传递，这只是说必须用 FAR 指针来做，而不是用与模型相关的指针（没有 far/near 标志）。这里有一个编译器选择项（加“w”到内存模型选择），它会在本地变量的地址作为一个模型相关指针传递时产生一个警告。使用此选择项，注意警告，并且把这些参数换为 FAR 指针。

2.6.2 DLL 和数据段

如果 DLL 同时被不同的应用程序使用，DLL 也只有一个数据段拷贝。这意味着 DLL 需要保存每一个应用程序的信息，开发者需要为此做些专门的工作。例如 DLL 可以把每个应用程序的所有信息封装到一个 AppInstance 的结构中，并且为每个使用 DLL 的应用程序分配一个新结构。接着在每个入口，DLL 就会标示出哪个应用程序调用了它，并且确定相应的 AppInstance 结构。

本书将不再深入讲述这个问题，因为在驱动程序 DLL 中不存在这个问题。通常一个驱动程序 DLL 为多个应用程序提供了一个设备的串行访问方法。也就是说，如果应用程序 1 正在通过 DLL 访问设备，则应用程序 2 就不能访问设备。另一方面，应用程序 1 可以使用设备 1，并且应用程序 2 可以使用设备 2，两个设备都由一个 DLL 来管理。但是只能作为每个设备的数据被管理，而不是每个应用程序管理的问题。本书将在第 4 章中详细讲解有关设备数据的问题。

2.6.3 DLL 和动态分配内存的主权

一个 Windows 的 DLL 在动态分配内存时一定要小心，因为这个内存可能属于其他正在运行的应用程序或 DLL 自己，完全依赖于内存分配。动态分配内存可以通过使用 GlobalAlloc ( )的调用来完成内存管理器，即完成内存的分配，也可以通过 C 语言的运行期函数 malloc ( )来实现。

当 DLL 直接调用 GlobalAlloc ( )时，DLL 确定的内存由 DLL 或调用程序所拥有。如果 DLL 在调用 GlobalAlloc ( )时使用 GMEM_SHARE，则 DLL 拥有内存；如果不是，则应用程序拥有内存（注：将在以后说明 GlobalAlloc ( )使用的参数）。对于 Microsoft C v7.0 及以后的版本，C 运行期库中的函数 malloc ( )在 DLL 被调用时一直使用 GMEM_SHARE 标志，所以 DLL 拥有动态分配的缓冲区。

应用程序或 DLL 都可以通过调用 GlobalFree ( )或 Free ( )来直接释放缓冲区。如果没有人直接释放动态分配内存，那么所有权的问题就变得重要了：如果应用程序拥有 DLL 所分配的内存，那么内存将在应用程序结束时被 Windows 自动释放；如果 DLL 自己拥有

内存，则内存只有在 DLL 卸载时才被 Windows 释放，在所有使用 DLL 的应用程序都退出后才会发生。

那么，谁拥有驱动程序分配的缓冲区呢？是应用程序还是 DLL？如果不是中断方式的驱动程序，这不是问题。在这种情况下，DLL 只是作为被应用程序调用的结果来执行。如果应用程序终止了，正常退出或者崩溃，它不会再调用 DLL，DLL 将不再使用缓冲区。

对于一个控制中断的驱动程序，在中断时所使用的动态分配缓冲区将由 DLL 拥有。要理解为什么这样，我们需要看看由应用程序拥有缓冲区时会发生什么情况。假设一个驱动程序有一个 Open ( )入口点和一个 Close ( )入口点.在 Open ( )期间，有一个缓冲区被分配，并有一个中断处理器被安装。在 Close ( )时，缓冲区被释放，处理器被去掉。现在假设一个应用程序在退出前崩溃或遗漏了 Close ( )命令，当应用程序退出时，Windows 自己释放分配的内存。如果此时你访问驱动程序的内存，糟糕的事情就发生了，因为内存已经被释放掉。相反，如果驱动程序自己拥有缓冲区，中断处理器就能继续安全地访问缓冲区。

2.6.4 DLL 的初始化和终止

一个 Windows DLL 可以通过调用 LibMain ( )来包含一个专用的初始化入口点，并通过调用 WEP ( )包含一个专用的终止入口点(为 Windows 退出程序)。如果存在，那么 LibMain ( )在 DLL 加载时被调用；如果 DLL 包含 WEP ( )函数，则它在 DLL 卸载时被调用。

对于许多 DLL，LibMain ( )中的初始化代码所做的事，与注册窗口类以及初始化局部堆阵（位于 DLL 的数据段中）相似。然而驱动程序 DLL 不注册窗口，并且一般使用全局堆阵来代替局部堆阵（局部堆阵太小）。本书中的驱动程序 DLL 不包含 LibMain ( )，相反，也可以在另一个使用 DLL 的应用程序中直接调用入口点，使用 Open ( )命令来进行初始化。类似地，大多数 DLL 使用 WEP ( )入口点来取消窗口类的注册。驱动程序 DLL 没有窗口类，而本书中的 DLL 在入口点 Close ( )中做清除工作。

2.7 DLL 的函数要求

虽然 DLL 可以以几种内存模式进行编译，但所有输出到应用程序的 DLL 函数都必须声明为 far。原因很简单，一个 DLL 拥有自己和应用程序分离的代码段和数据段。如果你把自己的 DLL 作为中等模型或者大模型来编译，那么根据定义，所有的函数都是 far；如果你把 DLL 作为小模型来编译，就必须用关键字 far 明确声明 DLL 的入口点。在本书中，选择用小模型对驱动程序 DLL 进行编译（其原因在第 4 章中解释），所以这些 DLL 的所有入口点都被声明为 far。

DLL 的入口点还用关键字_export 来声明。它告诉连接器为这些函数产生专门的入口定义记录，加载器在运行时使用它们来决定对 DLL 的查询。

一个输出的 DLL 函数必须在函数的开始和结束处包含一个特殊的代码部分。开始部分的代码叫做序言进程，它的目的是“准备”DS 寄存器，意思是它必须与 DLL 的数据段

同时读入 DS。这是必须的，因为进入 DLL 时，DS 包含着正在调用应用程序的数据段，与 DLL 的数据段不同。结束部分的代码叫做收尾进程，它存储着到调用者原始数据段 DS 的值。

一个特殊的编译器选项告诉编译器哪个函数需要序言进程和收尾进程代码。在 MSC v8.0 中，/GD 标志告诉编译器所有声明为_export 的函数产生序言进程/收尾进程。本书中 DLL 的制造文件（makefile）使用/GD 标志。

2.8 框架驱动程序

给出的第一个驱动程序的例子是一个框架或样板驱动程序（见程序例 2.1~2.3）。它并不和某一硬件相连，但有一组输出函数（一般来说，这些函数对多数类型的驱动程序都足够用）OpenDevice ( ), CloseDevice ( ), WriteDevice ( ), ReadDevice ( ), GetWriteStatus ( ), GetRead Status ( ), GetDriverParameters ( ), SetDriverParameters ( )和 GetDriverCapabilities ( )。正如那些例子，当驱动程序变得更复杂时，就会在框架驱动程序的函数中逐个地加入功能。当然，你的驱动程序可能需要更多的与这些函数不同的功能，那时可以任意加入你所需的函数，因为 Windows 不会对非标准设备的驱动程序接口加以限定。

程序例 2.1

框架驱动程序的头文件（SKELETON.H）

```
typedef  struct
{
  word  usDevNumber;
  LPBYTE  lpReadBuffer;
}  DEVICECONTEXT, FAR *HDEVICE;

typedef  struct
{
  word  usReadBufSize;
}  DRIVERPARAMS, FAR  *PDRIVERPARAMS;
typedef  struct
{
  word  version;
}  DRIVERCAPS, FAR  *PPDRIVERCAPS;
typedef  PDRIVERCAPS  FAR  *PPDRIVERCAPS;
HDEVICE  FAR  PASCAL  DeviceOpen(HWND  hwnd);
int  FAR  PASCAL  DeviceClose(HDEVICE);
int  FAR  PASCAL  DeviceGetWriteStatus(HDEVICE, LPWORD  pusStatus);
int  FAR  PASCAL  DeviceGetReadStatus(HDEVICE, LPWORD  pusStatus);
int  FAR  PASCAL  DeviceWrite(HDEVICE, LPBYTE  lpData, LPWORD  pcBytes);
int  FAR  PASCAL  DeviceRead(HDEVICE, LPBYTE  lpData, LPWORD  pcBytes);
int  FAR  PASCAL  DeviceSetParams(HDEVICE, PDRIVERPARAMS  pParams);
```

```
int FAR PASCAL DeviceGetParams(HDEVICE, PDRIVERPARAMS pParams);
int FAR PASCAL DeviceGetDriverCapabilites(HDEVICE, PPDRIVERCAPS
ppDriverCaps);
/*End of file /*
```

程序例 2.2
框架驱动程序的源文件（SKELETON.C）

```
#include <windows>
#include "skeleton.h"
DEVICECONTEXT Device1={0};
DRIVERPARAMS DefaultParams={1024};
HDEVICE FAR PASCAL _export DeviceOpen(HWND hwnd)
{
  OutputDebugStrig("DeviceOpen\n");
  return &Device1;
}
int FAR PASCAL _export DeviceClose(HDEVICE hDevice)
{
  OutputDebugStrig("DeviceClose\n");
  return 0;
}
int FAR PASCAL _export DeviceGetWriteStatus(HDEVICE hDevice, LPWORD
pusStatus)
{
  OutputDebugStrig("DeviceWriteStatus\n");
  return 0;
}
int FAR PASCAL _export DeviceGetReadStatus(HDEVICE hDevice, LPWORD
pusStatus)
{
  OutputDebugStrig("DeviceReadStatus\n");
  return 0;
}
int FAR PASCAL _export DeviceWrite(HDEVICE hDevice, LPBYTE lpData,
LPWORD pcBytes)
{
  OutputDebugStrig("DeviceWrite\n");
  return 0;
}
int FAR PASCAL _export DeviceRead(HDEVICE hDevice, LPBYTE lpData,
LPWORD pcBytes)
{
```

```
  OutputDebugStrig("DeviceRead\n");
  return 0;
}
int FAR PASCAL _export DeviceSetParams(HDEVICE hDevice,
PDRIVERPARAMS pParams)
{
  OutputDebugStrig("DeviceSetParams\n");
  return 0;
}
int FAR PASCAL _export DeviceGetParams(HDEVICE hDevice,
PDRIVERPARAMS pParams)
{
  OutputDebugStrig("DeviceGetParams\n");
  return 0;
}
int FAR PASCAL _export DeviceGetDriverCapabilites(HDEVICE hDevice,
PPDRIVERCAPS ppDriverCaps)
{
  OutputDebugStrig("DeviceGetDriverCapabilites\n");
  return 0;
}
/*End ot fils*/
```

程序例 2.3

框架驱动程序的定义文件（SKELETON.DEF）

```
LIBRARY        DRIVER
DESCRYPTION    "Skeleton Driver"
EXETYPE        WINDOWS
DATA           PRELOAD  MOVEABLE  SINGLE
CODE           PRELOAD  MOVEABLE  DISCARDABLE
```

框架驱动程序的每个函数只输出一个包含着函数名的跟踪消息。驱动程序通过 Windows API 函数 OutputDebugString ( )来输出这些消息。OutputDebugString ( )只使用一个字符串作为参数，但还可以使用更强大的函数 DebugOutput ( )。DebugOutput ( )使用一个格式字符串和一个数字变量参数，和 sprintf( )一样，提供更多有用的格式。Windows 重定向这些字符串到 AUX 设备（串口），你可以使用工具 DBWIN 把它们显示到一个窗口内（以后再深入介绍 DBWIN）。

2.9　建立框架驱动程序

建立驱动程序的步骤包括：

- 编译 .C 文件
- 连接成 .DLL 文件
- 运行资源编译器
- 创建入口库

为了能自动正确地完成建立驱动程序所需的步骤，使用 NMAKE 和一个制造文件（见程序例 2.4）。如果把制造文件和源文件拷贝到当前目录的话，只需键入 nmake-fskeleton.mak 就能建立框架驱动程序。

程序例 2.4

在 NMAKE 连接中使用的制造文件（SKELETON.MAK）

```
all : skeleton.dll

#DRIVER  DLL

skeleton.obj: skeleton.c  skeleton.h
  cl  -c  -W3  -ASw  -GD2s  -Oi  -Fc  $*.c

skeleton.dll : skeleton.def  skeleton.obj
  link  skeleton, skeleton.dll, skeleton.map /MAP, sdllcew libw /nod/noe,
skeleton.def
  implib  skeleton.lib  skeleton.dll
  copy  skeleton.dll  \windows\driver.dll
```

选择合适的编译器选项，对正确生成 Windows DLL 很关键。框架 DLL 使用的选项以及是否所有的 DLL 都应该遵循的选项标示，都列于表 2-1 中。

表 2-1 框架 DLL 使用的选项

项	要求	描述
c	强制的	只编译（不连接）
GD	强制的	产生序言进程来准备 DS
AS	可选的	小模式
Gs	强制的	禁止栈探测
Aw	推荐的	当 DLL 使用 near 指针来获取局部变量的地址时产生警告
W3	推荐的	警告级别 3（最高为 4）
G2	推荐的	产生代码为 286+（加速执行）
Zi	可选的	产生 CodeView 调试信息
Oi	可选的	使用固有命令(比 inp/outp/strcpy/memcpy 更快)
Fc	可选的	产生汇编输出

一个 Windows DLL 的连接过程和建立 DOS 应用程序的过程相似，除了在连接时都必须指定一个.DEF 文件外，这个.DEF 文件必须包括语句 EXETYPE WINDOWS 和

LIBRARY。用两个语句告诉连接器建立一个 Windows DLL。其他必须指定的 DEF 语句还包括 CODE 和 DATA，决定 DLL 的数据段和代码段的属性。在以后的内容中将讨论这些属性。

一个 DOS 应用程序的连接器命令行一般不指定库文件，没有必要指定，因为 C 编译器把信息嵌入到.OBJ 文件中，告诉连接器引入哪个库（小型、中等或大型）。Windows DLL 需要一个 C 库的专用版本（?dllcew.lib 代替?libcew.lib），这里“?”是内存模式的简写。当使用 MSC v8.0 时，应当使用/NOD 选项以便连接器不会调入在.OBJ 文件中命名的 C 库。另外，还应该指定 LIBW 作为二级库。

接着使用 IMPLIB 来生成 DLL 的入口库。IMPLIB 使用 DLL 的.DEF 文件作为输入，并且生成一个包含出口函数的名字和模块的.LIB 文件，然后将这个.LIB 文件作为一个库连接到使用 DLL 的应用程序上。IMPLIB 不需要任何选项开关。

制造文件中的最后一步是拷贝驱动程序 DLL 到 Windows 目录下（这不是必须的，但却是有用的）。当运行时，Windows 使用与定位 Windows 文件一样的方法来定位一个 DLL：搜索当前目录、Windows 目录、Windows 系统目录以及在 PATH 中所列的目录。通过拷贝驱动程序到 Windows 的目录下，就可以使用应用程序（以及 DLL）而不必考虑当前的目录和 PATH 变量。

2.10 DLL 需要应用程序

Windows DLL 自己不能运行，它必须被 Windows 程序调用，所以需要一个调用驱动程序 DLL 的 Windows 应用程序。为此，将使用 Windows 的应用程序 TESTDRIV.EXE。

TESTDRIV 是一个简单的 Windows 程序。它的界面是一个包含几个菜单项的简单菜单，每个子菜单对应一个驱动程序函数：

OpenDevice ( )，CloseDevice ( )，WriteDevice ( )，ReadDevice ( )，GetWriteStatus ( )，GetReadStatus ( )，GetDriverParameters ( )，SetDriverParameters ( )以及 GetDriverCapabilities ( )。选择一个菜单项，TESTDRIV 就会调用驱动程序中相应的函数（如图 2-1）。

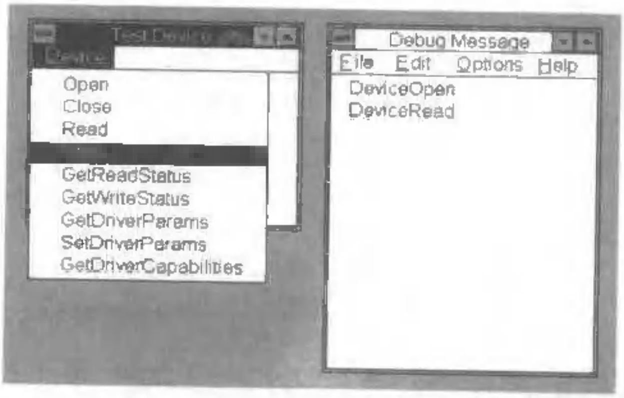

图 2-1　Windows 应用程序 TESTDRIV.EXE

以上产生的结果来自于所选择的函数。一些函数（如 Read ( )和 Write ( )）需要用户提供输入参数，所以在调用 Write ( )前，TESTDRIV 提示用户给出 Write ( )所需的数据及其长度。同样在调用 Read ( )前，会提示所需数据的长度（TESTDRIV 提供自己的数据缓冲区），当采集到输入参数后，TESTDRIV 调用驱动程序的函数。

如果驱动程序函数返回一个错误码，TESTDRIV 将显示一个消息框。如果函数是以下 3 个带有输出参数的一个(GetWriteStatus ( )，GetReadStatus ( )或 GetDriverCapabilities ( ))，则输出参数将在消息框中显示出来。

因为以后章节中的驱动程序的例子将保持同一组出口函数，所以可以使用 TESTDRIV 来测试所有的驱动程序。由于驱动程序函数动态地（而非静态）连接到应用程序，在每次修改驱动程序后不需要再重新连接 TESTDRIV。

2.11 驱动程序的调试工具

使用 TESTDRIV 测试驱动程序时，运行 Windows 的应用程序 DBWIN.EXE 也很有用，它是 Windows SDK 中的一个工具（SDK 已被群组件如 VC++PE 所取代）。DBWIN.EXE 捕获 Windows 的应用程序或在 DLL 中调用 OutputDebugString ( )和 BebugOutput ( )输出的字符串。DBWIN.EXE 可以重定向字符串到客户窗口、二级监视器或者串口。通过增加更多的 OutputDebugString ( )到驱动程序，可以跟踪它的执行路径，这样就可以进行基本的调试。

当然，这代替不了对真正调试器的需要。应用级的调试器（如 CodeView for Windows）可以用来调试许多类型的驱动程序，CodeView 适合对它们的调试，但不适合对控制中断驱动程序和 VxD 的调试。对此，应该使用 Windows SDK 提供的 WDEB386 或者由 NuMega 公司提供的 SoftIce/Windows。二者提供几乎相同的功能，但 SoftIce 改进了 WDEB386 的两点不足。它不需要通过串口连接的独立调试终端（WDEB386 需要），而可以在源级调试 C 代码。

2.12 总　　结

通过本章的讲解，建立一个开发驱动程序的测试环境，并进一步说明使用一定的工具，正确建立连接到应用程序的 DLL。由于本章的框架驱动程序并不做什么，可以由它来验证到和应用程序间的互通，这为制作实际控制硬件的 DLL（第 3 章的主题）作好了准备。

第 3 章　与硬件相连接

我们都知道，在 DOS 系统中，允许编程者在某些时候直接控制设备。而在 Windows 系统中，则不允许编程者直接与硬件设备打交道，这是由于该系统对物理机器资源具有保护特性。在一个完善、高性能的驱动程序中，Windows 的保护机制使设备访问变得非常复杂。不过简单的查询方式驱动程序对设备访问仍然十分直接。本章要讲述的是怎样把第 2 章的框架驱动程序，从没有实际的操作转换成一个完全、简单、并能实际控制物理设备的查询方式的驱动程序。在 PC 结构中，一个驱动程序可以被设计成内存映射或端口映射的输入/输出（I/O）。本章还讲解了与这两种访问类型相关的一些基本问题。首先通过给出一个完整的查询方式——串口驱动程序来说明简单的端口映射，接着再说明怎样修改端口映射的程序来实现对一个虚构的内存映射设备的访问。

3.1　端口映射和内存映射的比较

一个 PC 上的硬件设备可以被定位于两种独立的地址空间，这两种地址空间分别是内存地址和 I/O 地址。一个硬件设备的地址在内存地址空间的范围称为内存映射，它可以像内存一样准确地被访问，也可以通过一些内存访问指令（如 MOV，ADD，OR 等）来访问。而一个硬件设备的地址在 I/O 地址空间的范围称为 I/O 映射，在 I/O 地址空间的设备（I/O 映射）只能通过专用的 I/O 指令来访问，如 IN，OUT 以及由它们派生出的命令。这两个地址空间的范围是完全不同的。另一个不同是 I / O 地址空间比内存地址空间小得多。内存地址空间的大小范围从 8086 的 1MB 到 80386 的 4GB，目前还更多；而 I / O 地址空间对于所有的 80x86 处理器只有 64KB。

3.1.1　访问端口映射硬件

大部分高级语言都是通过库函数的形式来支持 IN 和 OUT 命令，例如 INP ( )或_INP ( )，OUTP ( )或_OUTP ( )等。虽然这些库的 Windows 标准版去掉了一些函数，但 INP ( )和 OUTP ( )都仍然保留着。大部分的编译器都能通过 inlining 来执行一个优化命令，如在调用 INP ( )/OUTP ( )库命令的地方可以直接用简单的 IN/OUT 命令来代替，使调用命令简单化。这显然是更有效的处理方法，特别是当驱动程序中有大量的输入或输出要处理时，这种方法的优点就显得更为重要。

3.1.2　访问内存映射硬件

最早的一段时间，PC 系统的设计者们有意在处理机的地址空间 A0000H 至 F0000H 之

间留下了段空地址。在这些位置上没有 RAM 存在，留下它们供内存映射设备使用。在 DOS 系统下，访问内存映射设备需要生成一个指向某一位置的指针，并间接引用指针，就能访问到所需要的地址段。在 Windows 下访问设备的方法基本一样，但当 Windows 在增强模式下运行时，其过程十分复杂。处理器使用了两个特殊的地址级别转换，把一个指针转变成内存映射设备能够识别的地址。了解产生正确访问内存映射设备的指针地址转换机制是必要的。

3.1.3　两步骤地址转换过程

在 Windows 应用程序中，每个地址包括两部分：选择器和偏移量。这种形式的地址被称为逻辑地址，也叫虚拟地址。我们在编程中使用的是逻辑地址，当一个 Windows 应用程序查询地址时，处理器将逻辑地址（选择器/偏移量的组合）转换成一种叫做线性地址的直接形式，接着再转换成物理地址，物理地址直接连到总线上并被设备译码。由 Windows 应用程序所控制的逻辑地址和相应的物理地址之间没有数学关系。它们的转换步骤如下：

（1）从逻辑地址到线性地址

在逻辑地址（选择器/偏移量）中，使用的选择器只是一个描述符表的索引，每个描述符表保存一个段的基地址。处理器是这样进行第一步逻辑到线性转换的：通过相应的选择器查寻出基地址，并把基地址与偏移量相加，所得结果即为线性地址。这一过程在图 3-1 中进行了描述。

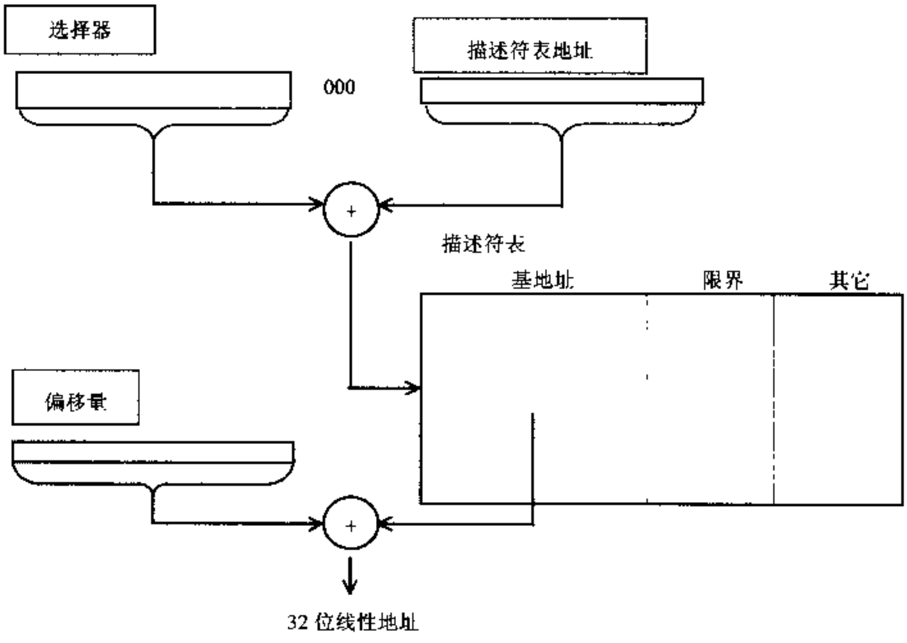

图 3-1　逻辑地址到线性地址的转换过程

（2）从线性地址到物理地址

转换的另一步是从线性地址到物理地址，包括另一套数据结构：页表以及页表目录。这些结构把线性地址空间的每 4KB 的“页”一起映射到物理内存上的 4KB 的页，或在 RAM 中，或在硬盘上（如图 3-2）。Windows 对页表到物理内存再变换的广泛使用，能够满足每个应用程序的各种需要。下一章将详细解释页机制，这里只要认识到在线性内存与物理内存间没有数学关系就足够了。只有操作系统能构造出确保指向给定的物理位置的指针，要访问一个内存映射设备，必须使得操作系统给出一个对应于物理内存页的选择器。每个段的基地址和限界（属于其他信息）存储在一个叫做描述符的 8 字节数据结构中，描述符按顺序存在描述符表中，而描述符表则由操作系统保存。选择器是到描述符表的索引。

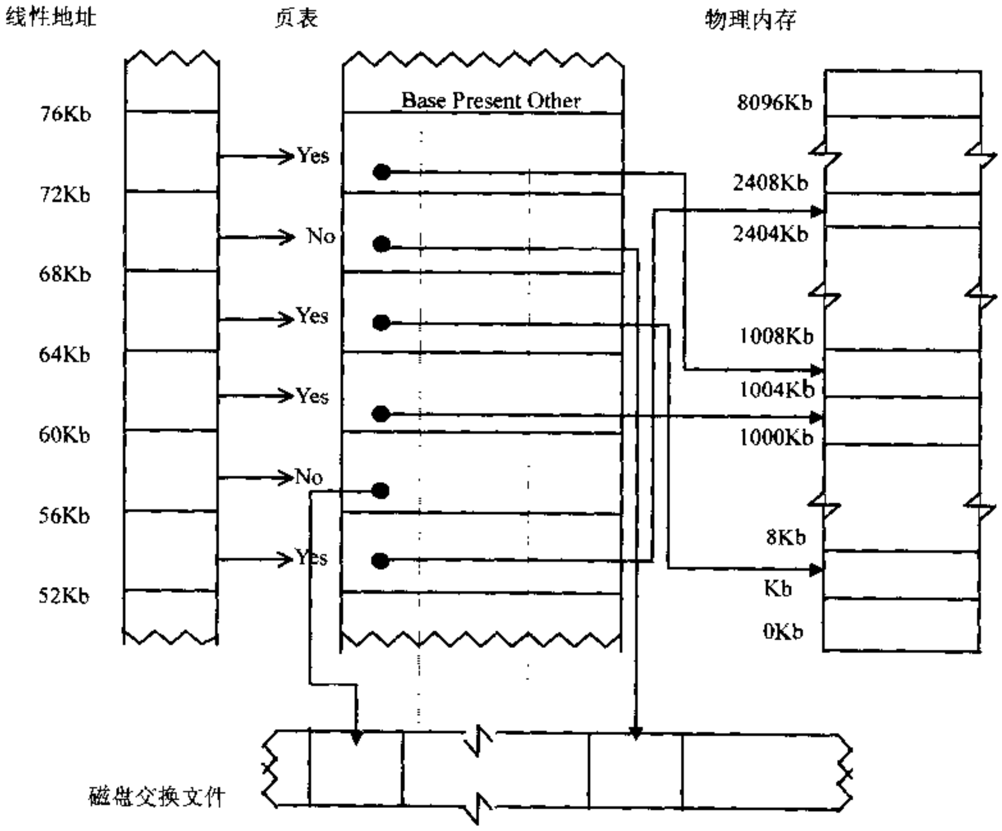

图 3-2　线性地址到物理地址的转换过程

80386 地址转换硬件使用两个描述符表：局部描述符表（LDT）和全局描述符表（GDT）。当编写一个非 VxD 驱动程序时，编程人员很少需要关心这一区别，因为用于非 VxD 驱动程序的所有选择器是 LDT 选择器，VxD 才对 LDT 和 GDT 都进行访问。我们将在第二部分讨论 VxD。

当一个段寄存器（CS，DS，SS 或 ES）被调入时，处理器就从主内存的描述符表中读入相关联的 8 字节描述符，并把它寄存在一个隐藏的描述符高速存储器中。每个段寄存器

都有一个描述符高速寄存器，并且像所有的寄存器一样，它们由处理器芯片自己设置（而非主内存）。当处理器执行一个包含对已调入段进行内存查询的命令时，使用存储在高速寄存器中的基地址、限界和特权信息以代替再次从主内存的描述符表中读取 8 字节。这些高速寄存器对适当处理器性能的级别要求很严格。

3.1.4　对 1MB 以下的设备映射使用预定义的选择器

如果一个设备映射到 A0000H~F0000H 段，在主机没有使用的物理地址空间中，就可以采用非常简单的办法使指针访问设备。Windows 分别为物理位置 A0000H，B0000H，C0000H，D0000H，E0000H，F0000H 提供了预分配的选择器。这些选择器实际上是从 Windows 系统的 DLL 中输出的变量，并且相应地命名为_A000H，_B000H 等等。Windows 已经建立了选择器的基地址以及对应的相关页表入口，选择器_B000H 实际上映射到物理地址 B0000H。每个选择器都有 64KB 的限制，所以_A000H 映射到 A0000H－AFFFFH，_B000H 映射到 B0000H－BFFFFH，如此类推。

为使你的设备生成一个指针，选择相应的选择器和偏移量，例如_D000H 和偏移量 8000H 组合指向在 D8000H 的设备。使用 MAKELP（_D000H，8000H）来把选择器/偏移量组合成为一个可用指针很简单，下面的代码段描述了所需的 3 步：

```
//KERNEL.__D000 中是双下划线
//在#define SelD000H(&MyD000H)中是单下划线
//访问在物理地址 D0000H 的内存映射适配器
//必须在你的.DEF 文件中加入选择器:
//IMPORTS
//   __MyD000H=KERNEL.__D000H
{
extern  WORD  _MyD000H;
#define  SelD000H(&_MyD000H)
char  far  *lpAdapter=MAKELP(SelD000H,0x8000);
  }
```

3.2　端口映射例子程序

在例子程序 3.1~3.4 中，包括一个简单的标准串口的查询方式驱动程序。利用串口能制成很好的例子程序，因为每个系统都有一个位于 I/O 空间的串口，并且 Windows 并不需要在启动时接管这个设备。我们可以很容易安装自己的串口处理器，而不会导致 Windows 的混淆，搞不清其他的 PC 标准设备，如键盘和计时器。第二部分将解释计算机的端口俘获机制，并详细解释 Windows 怎样“接管”设备。

3.2.1 例子程序 3.1

标准串口查询方式驱动程序例子的头文件（POLLED.H）

```
typedef struct
{
  WORD usReadBufSize;
}DRIVERPARAMS, FAR *PDRIVERPARAMS;

typedef struct
{
  WORD version;
}DRIVERCAPS, FAR *PDRIVERCAPS;

typedef PDRIVERCAPS FAR *PPDRIVERCAPS;

typedef struct
{
  WORD usDevNumber;
  WORD usIoBase;
  BYTE bFlags;
  DRIVERPARAMS params;
}DEVICECONTEXT, FAR *HDEVICE;

HDEVICE FAR PASCAL DeviceOpen(HWND hwnd);
int FAR PASCAL DeviceClose(HDEVICE);
int FAR PASCAL DeviceGetWriteStatus(HDEVICE, LPWORD pusStatus);
int FAR PASCAL DeviceGetReadStatus(HDEVICE, LPWORD pusStatus);
int FAR PASCAL DeviceWrite(HDEVICE, LPBYTE lpData, LPWORD pcBytes);
int FAR PASCAL DeviceRead(HDEVICE, LPBYTE lpData, LPWORD pcBytes);
int FAR PASCAL DeviceSetParams(HDEVICE, PDRIVERPARAMS pParams);
int FAR PASCAL DeviceGetParams(HDEVICE, PDRIVERPARAMS pParams);
int  FAR  PASCAL  DeviceGetDriverCapabilites(HDEVICE, PPDRIVERCAPS
ppDriverCaps);
/*End of file /*
```

3.2.2 例子程序 3.2

标准串口的头文件（UART.H）

```
#define  UART REG THR     0X00
```

```
#define  UART_REG_RDR     0X00
#define  UART_REG_IER     0X01
#define  UART_REG_IIR     0X02
#define  UART_REG_LCR     0X03
#define  UART_REG_MCR     0X04
#define  UART_REG_LSR     0X05
#define  UART_REG_BAUDLO  0X00
#define  UART_REG_BAUDHI  0X01

#define  UART_IIR_NONE    0X01
#define  UART_IIR_THRE    0X02
#define  UART_IIR_RXRDY   0X04
#define  UART_IER_THRE    0X02
#define  UART_IER_RXRDY   0X01
#define  UART_MCR_OUT2    0X08
#define  UART_MCR_LOOP    0X10
#define  UART_LSR_THRE    0X20
#define  UART_LCR_DLAB    0X80
#define  UART_LCR_8N1     0X03
#define  UART_ISR_RXRDY   0X01
#define  BAUD_1200        0X60

//End  Of  File
```

3.2.3　例子程序 3.3

标准串口查询方式驱动程序例子的源文件代码（POLLED.C）

```
#include <windows>
#include <conio.h>
#include "polled.h"
#include "uart.h"

#define FLAGS_OPEN    0x04

#define SET(value,mask)     (value|=mask)
#define GET(value,mask)     (value&=(~mask))

DEVICECONTEXT Device1={0,0x3F8,0,NULL};
DRIVERPARAMS DefaultParams={1024};
DRIVERCAPS={0x0103};
```

```
HDEVICE FAR PASCAL _export DeviceOpen(HWND hwnd)
{
  OutputDebugStrig("DeviceOpen\n");

  hDevice=&Device1;
  if(hDevice->bFlags&FLAGS_OPEN)
      return (HDEVICE)-1;

  hDevice->params=DefaultParams;

  //Configure  UART
  outp(hDevice->usIoBase+UART_REG_IER,0);
  outp(hDevice->usIoBase+UART_REG_LCR,UART_LCR_DLAB);
  outp(hDevice->usIoBase+UART_REG_BAUDLO,BAUD_1200);
  outp(hDevice->usIoBase+UART_REG_BAUDHI,0);
  outp(hDevice->usIoBase+UART_REG_LCR,UART_LCR_8N1);
  outp(hDevice->usIoBase+UART_REG_MCR,UART_MCR_LOOP);
  inp(hDevice->usIoBase+UART_REG_LSR);
  inp(hDevice->usIoBase+UART_REG_RDR);
  SET(hDevice->bFlags,FLAGS_OPEN);

  return hDevice;
}

int FAR PASCAL _export DeviceClose(HDEVICE hDevice)
{
  OutputDebugStrig("DeviceClose\n");

  if(!ValidHanddle(hDevice))
      return -1;
  if((hDevice->bFlags&FLAGS_OPEN)==0)
      return -1;

  CLR(hDevice->bFlags,FLAG_OPEN);

  return 0;
}

int  FAR  PASCAL  _export  DeviceGetWriteStatus(HDEVICE  hDevice,LPWORD
pusStatus)
{
  OutputDebugStrig("DeviceWriteStatus\n");
```

```
  if(!ValidHanddle(hDevicc))
      return -1;
  if((hDevice->bFlags&FLAGS_OPEN)==0)
      return -1;

  if(inp(hDevice->usIoBase+UART_REG_LSR)&UART_LSR_THRE)
      *pusStatus=1;                 //ready to transmit
  else
      *pusStatus=0;                 //not ready to transmit

  return 0;
}

int FAR PASCAL _export DeviceGetReadStatus(HDEVICE hDevice,LPWORD
pusStatus)
{
  OutputDebugStrig("DeviceReadStatus\n");

  if(!ValidHanddle(hDevice))
      return -1;
  if((hDevice->bFlags&FLAGS_OPEN)==0)
      return -1;

  if(inp(hDevice->usIoBase+UART_REG_LSR)&UART_LSR_RXRDY)
      *pusStatus=1;                 //data ready
  else
      *pusStatus=0;                 //no data rady

  return 0;
}

int FAR PASCAL _export DeviceWrite(HDEVICE hDevice,LPBYTE
lpData,LPWORD pcBytes)
{
  WORD i;
  OutputDebugStrig("DeviceWrite\n");

  if(!lpData)
      return -1;
  if(!ValidHanddle(hDevice))
      return -1;
```

```
  if((hDevice->bFlags&FLAGS_OPEN)==0)
      return -1;

  for(i=0; i<*pcBytes; i++)
  {
      while((inp(hDevice->usIoBase+UART_REG_LSR)&UART_LSR_THRE)==0)
          outp(hDevice->usIoBase+UART_REG_THR,lpData[i]);
  }

  return 0;
}

int  FAR  PASCAL  _export  DeviceRead(HDEVICE  hDevice,LPBYTE
lpData,LPWORD  pcBytes)
{
  WORD i;
  OutputDebugStrig("DeviceRead\n");

  if(!lpData)
      return -1;
  if(!ValidHanddle(hDevice))
      return -1;
  if((hDevice->bFlags&FLAGS_OPEN)==0)
      return -1;

  for(i=0; i<*pcBytes; i++)
  {
      while((inp(hDevice->usIoBase+UART_REG_LSR)&UART_LSR_RXRDY)==0)
          lpData[i]=INp(hDevice->usIoBase+UART_REG_THR);
  }

  return 0;
}

int  FAR  PASCAL  _export  DeviceSetParams(HDEVICE
hDevice,PDRIVERPARAMS  pParams)
{
  OutputDebugStrig("DeviceSetParams\n");

  if(!pParams)
      return -1;
  if(!ValidHanddle(hDevice))
```

```
        return -1;
    if((hDevice->bFlags&FLAGS_OPEN)==0)
        return -1;
hDevice->params=*pParams;

    return 0;
}

int  FAR  PASCAL  _export  DeviceGetParams(HDEVICE
hDevice,PDRIVERPARAMS  pParams)
{
  OutputDebugStrig("DeviceGetParams\n");

  if(!pParams)
      return -1;
  if(!ValidHanddle(hDevice))
      return -1;
  if((hDevice->bFlags&FLAGS_OPEN)==0)
      return -1;
  *pParams=hDevice->params;
  return 0;
}

int FAR PASCAL _export DeviceGetDriverCapabilites(HDEVICE hDevice,
         PPDRIVERCAPS ppDriverCaps)
{
  OutputDebugStrig("DeviceGetDriverCapabilites\n");

  if(!ppDriverCaps)
      return -1;
  if(!ValidHanddle(hDevice))
      return -1;
  if((hDevice->bFlags&FLAGS_OPEN)==0)
      return -1;
  *ppDriverCaps=&DriverCaps;

  return 0;
}

BOOL ValidHandle(HDEVICE hDevice)
{
  return (hDevice==&Device1);
}
/*End of fils*/
```

这个驱动程序输出到与第 2 章介绍的框架程序相同的公用界面，但是这个版本的服务

命令不仅仅是一个框架。在 skeleton 的程序中，所有的驱动程序函数都在 DLL 中。这并不意味着这是一个高性能、有商用品质的驱动程序，它只是能很容易地用于支持多个口或者不同的通用异步收发器（UARTS）。由于重点是描述广泛应用的驱动程序的技术和原理，所以我还会尽量避免陷入硬件的错综复杂和串口通信的细节中。（表 3-1 列出了 8250UART 的主要设置）。通过剖析控制器的最基本设置，可以把它们用到自己的硬件上。

表 3-1　8250 UART 的基本设置

偏移量	名　称	访　问
6	调制解调器状态	R
5	线路状态（LSR）	R
4	调制解调器控制	R/W
3	线路控制	R/W
2	中断识别	R
1	中断允许	R/W
0	接受数据	R
0	发送数据	W
1(当 LCR bit7=1)	波特率因子 MSB	R/W
0(当 LCR bit7=1)	波特率因子 LSB	R/W

3.2.4　例子程序 3.4

标准串口查询方式驱动程序例子的定义文件（POLLED.DEF）

```
LIBRARY      DRIVER
DESCRYPTION  "Polled Mode Driver"
EXETYPE      WINDOWS
DATA         PRELOAD  MOVEABLE  SINGLE
CODE         PRELOAD  MOVEABLE  DISCARDABLE
```

3.2.5　例子程序 3.5

标准串口查询方式驱动程序例子的制造文件（POLLED.MAK）

```
all: polled.dll

#DRIVER  DLL

polled.obj: polled.c  polled.h
  cl -c -W3 -ASw -GD2s -Oi -Fc $*.c
```

```
polled.dll: polled.def polled.obj
link  polled,polled.dll,polled.map/MAP,sdllcew libw/nod/noe,polled.def
implib  polled.lib  polled.dll
copy  polled.dll  \windows
```

3.3　驱动程序设计概述

在讨论本章的查询方式驱动程序前，我们先对本书中所有 DLL 驱动程序的一般性设计进行一个基本的概述。

所有 DLL 驱动程序的例子都使用一个“设备上下文”结构存储设备的一个简单事例的状态和寻址信息，尤其对串口 COM1。这是在设备驱动程序中十分常用的技术，它使得驱动程序很容易适应多个设备。上下文结构的地址用作对设备的"句柄'，此句柄通过调用 DeviceOpen ( )来返回，并且作为参数用于所有其他进入的驱动程序。设备上下文结构如图 3-3 所示。

```
type struct
{
  WORD  usDevNumber;
  WORD  usIoBase;
  BOOL  bFlags;
  LPBYTE  lpReadBuf;
}DEVICECONTEXT;
```

图 3-3　设备上下文结构

就像第 2 章讲的，一个 DLL 可以被多个应用程序调用。许多 Windows DLL 用专门的代码来处理它，在一个“事例数据”结构中存储所有到应用程序的专用上下文信息。这里给出的 DLL 驱动程序例子没有使用应用程序专用事例数据结构，原因很简单，驱动程序接口被设计成在某一时刻只有一个应用程序能使用设备。设备的上下文结构为了根据所有驱动程序 DLL 的需要而去找出所有相应的数据。

一些建立问题（SS!＝DS 的问题已在第 2 章描述过）可以通过把 DLL 作为大模型编译来消除。尽管如此，本书的例子驱动程序都被编译为小模型而非大模型。实际上，不管是上一章的框架驱动程序还是本章的查询方式驱动程序，如果作为大模型来编译，都会工作很好。但是第 4 章的中断驱动的驱动程序以及第三部分中更完善的中断驱动的驱动程序在设计时，必须以小模型工作，其根本原因是中断控制器必须调入它们自己的数据段。由于大模型 DLL 有多个数据段，所以它将使访问中断控制口中的数据变得复杂。虽然这个访问数据的问题实际上可以解决，但与保持驱动程序作为小模型却很相似。只有很少的驱动程序与小模型的 64KB 代码和 64KB 数据的限制有冲突。

3.4 驱动程序例子

本章并不详细讨论查询式驱动程序代码，因为驱动程序本身很小且简单，对被所有驱动程序入口点使用的数据结构和参数验证代码的探讨是主要内容。

在本例子中，功能字只编码驱动程序的版本号，许多复杂的驱动程序可以通过测试设备来决定它的专用功能。例如一个多模式扫描器驱动程序可以查询附加的驱动程序，并把最大的分辨率和颜色浓度存入功能结构中。这一信息可以控制其他驱动程序命令的行为，并且可以被相应的应用程序所调用。

例子驱动程序不使用 DRIVERPARAMS 结构。此外，更复杂的驱动程序可以提供几个配置选择，这些选择器能记录在参数结构中并被所有相关的命令查询。

当要求打开一个新设备时，DeviceOpen ( )命令初始化设备上下文结构，配置 UART（如图 3-4），接着置 FLAG_OPEN 位。驱动程序的其他代表会议可以检测 FLAG_OPEN 位以确定被需求的服务适合于指定的设备。本章的例子只在状态字段中使用 FLAG_OPEN 位，以后的例子在此将记录附加的状态信息。

```
//Configure  UART
outp(hDevice->usIoBase+UART_REG_IER,0);
outp(hDevice->usIoBase+UART_REG_LCR,UART_LCR_DLAB);
outp(hDevice->usIoBase+UART_REG_BAUDLO,BAUD_1200);
outp(hDevice->usIoBase+UART_REG_BAUDHI,0);
outp(hDevice->usIoBase+UART_REG_LCR,UART_LCR_BN1);
outp(hDevice->usIoBase+UART_REG_MCR,UART_MCR_LOOP);
inp(hDevice->usIoBase+UART_REG_LSR);
inp(hDevice->usIoBase+UART_REG_RDR);

SET(hDevice->bFlags,FLAGS_OPEN);
```

图 3-4　打开一个新设备所使用的命令

为了使驱动程序坚固，每一命令都确认 hDevice 指针和设备当前状态。例如为了防止驱动程序试图处理一个不存在的设备结构，驱动程序入口点通过反复的测试来确认 hDevice 指针。

```
if(!ValidHandle(hDevice))
return  -1;
```

为了防止驱动程序被使用到没有打开的端口，驱动程序命令将测试 bFlags 字段。

```
if((hDevice->bFlags&FLAGS_OPEN)==0)
```

```
return -1;
```

DeviceRead ( )和 DeviceWrite ( )命令的接口类似于标准 C 库读和写。DeviceRead ( )命令需要一个连接设备上下文的句柄、一个指向数据缓冲区的指针（lpData）以及一个指向读取字节个数的指针（pcBytes）。接着一个条件循环把数据一次一个字节地从 UART 拷贝到缓冲区，直至达到确定个数的字节为止。DeviceWrite ( )命令的工作与它相似，但以相反的方向进行。它把一定个数的字节从缓冲区拷到 UART 的发送寄存器。图 3-5 给出了每个函数主要的条件循环。

```
FAR PASCAL DeviceWrite(HDEVICE hDevice,LPBYTE lpData,LPWORD
   WORD pcBytes)
{
WORD i;
//validity checks omitted for clarity
for(i=0; i<*pcBytes; i++)
{
while ((inp(hDevice->usIoBase+UART_REG_LSR)
  &UART_LSR_THRE)==0)
  outp(hDevice->usIoBase+UART_REG_THR,lpData[i]);
}
return 0;
}

WORD FAR PASCAL DeviceWrite(HDEVICE hDevice,LPBYTE
  lpData,LPWORD pcBytes)
{
WORD i;
//validity checks omitted for clarity
for(i=0; i<*pcBytes; i++)
{
while ((inp(hDevice->usIoBase+UART_REG_LSR)
 &UART_LSR_RXRDY)==0)
 lpData[i]=inp(hDevice->usIoBase+UART_REG_RDR);
}
return 0;
}
```

图 3-5　DeviceRead ( )和 DeviceWrite ( )函数中的主循环

似乎不需要为 DeviceRead ( )和 DeviceWrite ( )的调用提供指向所需字节个数的指针。这一接口对于查询方式驱动程序确定是多余的，因为所需字节的个数永远和被处理字节的个数相同。但这一复杂性却允许我们不用修改接器就可以很容易地支持第 4 章的中断驱动程序。保持同样的接口，说明第 2 章介绍的应用程序 TESTDRIV 可以工作于查询方式和中断方式驱动程序，而不用重新编译 TESTDRIV。

如果按照以下协定，也可以成功地在低速、全双工通信中使用这种类型的驱动程序。

- 每次读 / 写时只传输一个字节。
- 不要试图进行读，除非调用 DeviceGetReadStatus ( )表明有一个字节可用。

3.5　内存映射的方案

如果你以前没有遇到过一个真正的内存映射串口，你会发现它很容易适应例子驱动程序。假设一个相似的寄存器布局，不同之处主要是 DeviceOpen ( )中的一些代码，建立了一个指向设备的基地址指针，并用它代替 inp ( )和 outp ( )的调用来对内存进行查询。

假设端口被映射在地址 D8000H，那么下面的代码将建立基本指针（DEVICECONTEXT 结构已被调整，地址字段类型为 char far *）。

```
//IMPORTS(.DEF 文件中的项)
//   __MyD000H=KERNEL.__D000H
extern  WORD  _MyD000H;
#define  SelD000H(&_MyD000h)
DEVICECONTEXT Device1={0,MAKELP(SelD000H,0x8000),0,NULL}
而主循环将变为
for(i=0;  i<*pcBytes;  i++)
{
while  ((*(hDevice->usIoBase+UART_REG_LSR)&UART_LSR_RXRDY)==0)
  lpData[i]=*(hDevice->usIoBase+UART_REG_RDR);
}
```

3.6　先进内存的问题

许多内存映射设备只在 A0000H～EFFFFH 范围内占用远小于 64KB 的空间，然而设备可以占用更大的空间并且可被定位于高端内存（1MB 以上）。如果你想制造大于 64KB 或定位于高端内存的设备，就不能使用预构造的选择器。对于这些设备，需要调用 DOS 保护模式接口（DPMI）服务来建立合适的选择器。

DPMI 是一组服务，由 Windows 提供的通过 INT 31H 而非通常的 API 函数来访问。DPMI 对选择器的操作提供低级服务，控制中断向量表，在实模式和保护模式之间进行切换，并且控制页表。Windows 3.x 支持 DPMI V0.9。DPMI V1.0 也存在，但它不能被 Windows 3.x

所支持。

在扩展模式中，段不再被限定为64KB。Windows 函数 SetSelectorBase( )不允许大于64KB的限界，但 DPMI 的函数 SetSelectorLimit( )则允许。对于映射大于 64KB 的设备，你可以只创建一个选择器即可。然而棘手的是必须产生使用 32 位偏移量的代码。Windows 程序在 16 位代码段中运行，即内存查询使用 16 位偏移量。可以通过在每个命令前插入预安装字节来突破偏移量位数的限制，并强迫处理器使用 32 位偏移量。这需要在汇编中进行编码，如果你想移动大量字节，再大的努力也值得。

图 3-6 中的代码分配一个寻址大于 64KB 的内存适配器的选择器，可以使用这个选择器对整个范围清零。存储单元甚至可以使用 32 位命令一次清零 4 个字节（一个 DWORD）。这些代码完全是用汇编写的，因为它是在 Windows 3.X 下最容易产生 32 位命令的方法。

```
.MODEL  SMALL
.CODE
.386

zero32 PROC C PhysSize:DWORD, PhysBase:DWORD

mov  ax,0                ; DPMI Alloc Selector
mov  cx,1
int  31h

mov  bx,ax               ; Selector  from  Alloc
mov  cx,WORD PTR[PhysSize+2]
mov  dx,WORD PTR[PhysSize]
mov  ax,08h              ; DPMI Set Selector Limit
int  31h

mov  dx,bx               ; save selector
mov  bx,WORD PTR[PhysBase+2]
mov  cx,WORD PTR[PhysBase]
mov  si,WORD PTR[PhysSize+2]
mov  di,WORD PTR[PhysSize]
mov  ax,0800h            ; DPMI Map Physical Address
int  31h
push  dx                 ; save selector
mov  cx,bx               ; HI word of linear base
mov  dx,cx               ; LO word of linear base
pop  bx                  ; restore selector
```

```
mov  ax,07h             ; DPMI Set Selector Base
int  31h

mov  es,ba
xor  edi,edi
mov  ecx,PhysSize
shr  ecx,2
xor  eax,eax
rep  stosd es:[edi]

zero32  ENDP
END
```

图 3-6 使用一个选择器和 32 位命令对内存映射设备进行大范围的填充

3.7 设备映射大于 1MB 需要 DPMI 服务

虽然大部分内存映射设备定位于 A0000H 到 FFFFFH 之间，但也可以定位一个设备高于 FFFFFH（1MB）。RAM 一直都是连续映射在高于 1MB 的范围的，所以一个定位高于 FFFFFH 的设备必须被定位于超过物理内存的最后一个字节。另外，当 RAM 和设备都试图译码同一个物理地址时会产生硬件冲突，而且系统也不能正常运行。

生成指向内存映射设备的指针包括前面所讲的两步地址转换过程（逻辑到线性，线性到物理）中的全部建立步骤。第一步建选择器的基地址，可以使用 Windows API 选择器函数（AllocSelector ( )，SetSelectorBase ( )和 SetSelectorLimit ( )）。第二步建立页表入口以便选择器的基地址能映射到所希望的物理地址，没有建立页表的 Windows API 函数，但是 DPMI 提供的 MapPhisicalAddress ( )函数可供使用。

DPMI 的 MapPhisicalAddress ( )函数把物理地址作为输入，并且返回（通过页表）映射到物理地址的线性地址。弄明白这个 DPMI 的调用的使用，对于用不同的方法考虑两步地址转换过程很有帮助。假设我们希望有个指针指向物理地址 A0000H，由于页表的影响，分配选择器设置它们基地址为 A0000H，但不能保证选择器转换到物理地址 A0000H。注意，我们实际上并不关心线性地址是多少，只要它映射到物理地址 A0000H。

所以我们反过来，从物理地址 A0000H 开始。把它传给 DPMI 的函数 MapPhisical Address ( )，它回送一个线性地址，叫做 X。接着我们把这个线性地址交给 SetSelectorBase ( )。现在我们有了一个映射到线性地址 X 的选择器，它就映射到物理地址 A0000H。

这里还有一个细节没有讲到。页表中每页只有 4KB，所以大于 4KB 限界的选择器由多个页组成。每个页可以存在于物理内存中的任何地方，并且物理上不是相邻的。由于设备只能明白物理上相邻的内存，所以我们必须确保不仅第一页映射到 A0000H～A0FFFH，

而且下一个页也映射到 A1000H～A1FFFH，依此类推。实际上，DPMI 的 MapPhisical Address ( )能保证被映射的页是物理相连的，虽然这点在 DPMI 文件中并不明确。

图 3-7 给出了一个函数，它使用 Windows 选择器函数和 DPMI 的 MapPhisicalAddress ( ) 服务来获得一个指向定位于 1MB 以上的内存映射设备指针。实现这个函数需要以下 4 步：

- 分配一个选择器。
- 设置它的限界。
- 使用 DPMI 来获得与给定的物理地址相对应的线性地址。
- 对此线性地址设置选择器的基地址。

```
Void far *MapPhysToPtr(DWORD PhysBase,DWORD PhysSize)
{
  WORD myDs,sel;
  WORD HiBase,LoBase;
  _asm
  {
mov  cx,PhysBase
mov  bx,PhysBase+2
mov  di,PhysSize
mov  si,PhysSize+2
mov  ax,0800h      ; DPMI Map Physical Address
int  31h
mov  HiBase,bx
mov  LoBase,cx
  }
  //Set selector's linear address as given by DPMI Map Phys
  SetSelectorBase(sel,MAKELONG(LoBase,HiBase));
  retrun(MAKELP(sel,0));
}
```

图 3-7 获得指向定位于 1MB 以上的内存映射设备的指针函数

这段代码的唯一技巧是调用 AllocSelector ( )，需要一个参数——一个模型选择器。因为我们已经生成了一个访问数据（而非代码）的选择器，所以代码在 DS 的当前值中传递，把一个有效的数据选择器作为模型选择器参数。

图 3-7 中的函数有个限制，只能在小于 64KB 的空间工作。可以很容易地使它适应大于 64KB 的范围，用相应的 DPMI 中的选择器函数（支持大于 64KB）代替 Windows 选择器函数（不支持大于 64KB）。然后可以用前面所讲过的汇编语言代码和 32 位偏移量访问设备。

3.8 总　结

这个例子并不是一个完善的驱动程序，它的缺点是对 Windows 环境没有任何操作的仔细简化，例如完善的驱动程序应当测试接收机的过速错误和各种成帧错误。这些测试很容易加入，而不用关心 Windows 专有的执行问题。

查询方式的主要缺陷是直接影响全双工操作的执行。查询方式迫使应用程序一次只能处理一个字节，或者冒着丢失大量反方向中的数据的危险。对于一个单向的设备（例如一个哑打印机），这样的查询方式驱动程序的执行十分安全。对于固定的设备，Windows 设备驱动程序可能简单。对于一个双向设备（例如串口），需要提供可靠的双向通信，而不用应用程序的字节方式的管理，同时需要一个中断方式驱动程序。

Windows 把附加的执行限制强加于中断方式驱动程序，特别是内存缓冲区在应用程序和驱动程序间进行通信的程序。第 4 章将解释这些问题，并说明怎样把查询方式驱动程序转换成中断方式驱动程序。

第 4 章　中断方式设备驱动程序

本章将说明怎样建立一个中断方式的设备驱动程序。一个查询方式的驱动程序（例如第 3 章所讲的建立）十分简单，中断方式的版本比它稍微复杂一些，并具有明显的优点。中断方式驱动程序一般提供改善的吞吐量，在等待设备时不占有处理器。

一个 Windows 中断方式驱动程序的基本结构和 Dos 驱动程序的结构十分相似，包括一个服务于硬件器件的实时组件（中断服务程序或 ISR）和一个控制与应用程序或操作系统通信的更高级组件（指的是“驱动程序”）。驱动程序和 ISR 通常通过一个作为共享资源管理的缓冲区进行通信。

如果一个 Windows 设备驱动程序有一个常见的结构，它被 Windows 变为“虚拟化”服务，以保护基本的硬件资源，则在查询方式驱动查询中，这些虚拟服务对于编程人员来说几乎不可见。但在中断驱动环境中，这些虚拟的机制（特别是虚拟内存系统）变得更加可见。为了避免中断正在运行的应用程序（甚至是 Windows），首先应该明白这些虚拟服务程序是怎样工作的，以及编写相应的中断式驱动程序所必须遵循的协定。通过本章的学习，希望能够到达如下要求：

1．明白内存系统工作的原理，能创造中断的安全代码和数据结构。

2．知道当访问安全的中断缓冲区和数据结构时必须遵守的协定。

3．知道安装设备驱动程序时所必须遵守的协定。

本章的前半部分主要解释上述 3 项内容。本章的后半部分说明怎样用这些信息，把查询方式的驱动程序转换为一个基本的中断方式驱动程序。

4.1　Windows 虚拟内存

Windows 的内存管理器使用虚拟内存（也叫交换区域或换页）技术，使得程序能配置比系统中有效物理内存更大的空间。

内存管理器认为内存（更准确地叫做地址空间）是一系列的块，其中一些在存储器中，而其他一些在硬盘上。为了使物理内存能够接受新的内存命令，内存管理器可以做以下 3 件事中的任意一件。

1．把当前物理块中的内容交换到磁盘上。

2．丢弃当前物理块中的内容（这里的“丢弃”是指再次使用同样的物理内存，并要把前面存到磁盘上的东西覆盖，主要因为磁盘上已经存在了一个拷贝）。

3．移动当前内存中的内容，以建立一个相邻内存的更大的块。

以上的每个动作，对一个中断服务程序来说，都可能导致灾难性的结果。下面将解释以下问题：

1. Windows 怎样实现这些功能。
2. 为什么交换、丢弃或移动块会给中断驱动程序造成麻烦。
3. 怎样编写代码和分配数据才能使交换、丢弃和移动不发生。

4.2 实 现 方 法

虚拟内存需要用处理器硬件的一些帮助，特别是它的页特性，并且这些只能在扩展模式中使用。当可以分页时，选择器的基地址不是物理地址，而是被称为线性地址的中间过渡形式。这个线性地址通过使用内存管理器中的页表结构进行一次转换后才能作为物理地址被使用（如图 3-1 和图 3-2 所示）。

关于分页最有用的东西并不是转换的方法，而是标注一个特别、不存在的 4KB 的页。当内存变少时，内存管理器写（转送）一个页到磁盘，并在页表中作为不存在的页进行标注。如果后面的程序要访问在哪个页中的某个位置，“不存在”的标注将会使处理器产生一个页错误。这个错误（处理器异常）将暂停执行当前程序，并把控制和产生错误的页号，转移页错误处理。这个处理器（属于内存管理的一部分）使用页号来确定位于磁盘上的号，并把它读入内存。整个页处理过程由操作系统动态进行，处理产生对内存访问的过程是完全透明的。

如果页错误发生时，在物理内存中没有空闲的块，那么内存管理就会通过交换一个当前存在的页到磁盘上来产生空闲空间。

4.3 什么是可以丢弃的

Windows 应用程序组织到段的逻辑组件中，这是在 Intel80x86 处理器上运行的自然结果，因为它仍用分段内存模型。段可分为 3 种类型：代码段、数据段和资源段。代码段包括程序的代码，数据段包括一个程序的数据（包括栈和局部堆），资源段包括用户界面的资源，例如菜单、光标和位图等。

所有的段，不论是作为执行的一部分静态分配的，还是被运行的程序动态分配的，都是通过调用 Windows 内存管理器的 API 函数来分配。当程序被首次调入内存时，Windows 加载器以应用程序的多义来分配，产生调用分配的应用程序的代码、数据和资源被使用。当正在运行的程序需要附加的内存时，它就直接调用内存管理器的 API 函数。

每个段，不论是代码、数据或资源，都有一组可被内存管理器跟踪的属性，这些属性决定了内存管理器对段可以做什么，不可以做什么。内存管理器只能丢弃被标注为 discardable 的段。其他的属性包括 non-discardable fixed,noveable，swappable（也叫做 unpage-loclced)和 non- swappable(page-locked)。

注意： 这些特性可以用 4 种方法确定。作为连接过程的一部分被静态地确定，或作为对内存管理器的分配需要而动态地进行。程序的静态代码、数据和资源段的属性

在连接的时候被确定（在模块定义文件.DEF 中），接着加载器以应用程序的名义分配具有这些属性的段。由程序动态分配段的属性，通过直接调用内存管理器，作为函数调用的参数被确定。程序可能稍后再次通过调用内存管理器来改变段的属性。

4.4　为什么整理内存

当 Windows 调入程序资源时，它分段调入并且一个段一个段地顺序调入，把每个独立的段放入线性内存相邻的块中。这样，当调入一个段时，Windows 不仅必须有足够的空闲内存，而且必须有足够的相邻空闲内存，确切地说，必须有足够的线性相邻空闲内存。（线性内存与物理内存间的区别已在第 3 章介绍过）。如果空闲内存是非常破碎的，那么内存管理器就需要活动的块来生成更大的空闲块。

Windows 依靠处理器的保护模式可以很方便地移动内存。在保护模式下指针仍然是一个由段和偏移量组成的逻辑地址，但逻辑地址到物理地址的转换不只是简要的移位和相加的数学计算。在保护模式下的地址变换是一个分步操作的过程。首先，作为描述符表的索引，描述符存储在包含段的基地址的描述符表中。第二步类似于实模式的转换过程，偏移量加到基地址上。

注意： 当一个程序需要从内存管理器分配空间时，如果空闲内存被分散成许多小块而不是一个大块，那么即使有空闲内存也是不可用的，这个需求可能被拒绝。此问题被称为内存碎片。它甚至在 DOS 下也存在，但是多任务处理使这个问题变得更严重。Windows 内存管理器以及内存管理 API 函数的复杂性的存在，很大程度是为了消除这个问题，所以进一步介绍一下内存碎片是很有价值的。

图 4-1 描述了产生内存碎片的过程。开始时，所有的可用内存都位于一个大的堆（HEAP）

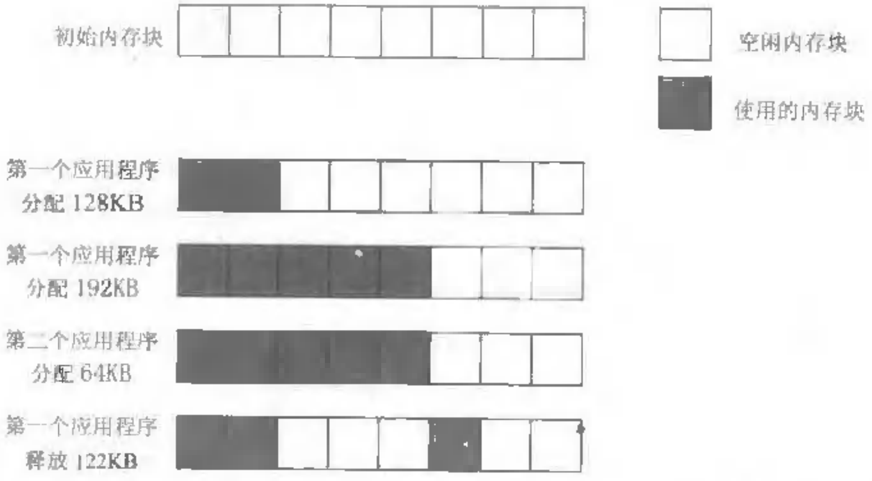

图 4-1　产生内存碎片的过程

的块内。接着第一个应用程序运行从堆中“切出”一块，然后这个应用程序又分配了第二个块。现在另一个应用程序开始运行并需要一个块，接着第一个应用程序解除第一块的分配。这样就在堆中留下一个洞，而现在的堆由两个空闲内存块组成。

这种在堆中有“洞”的情况就叫做内存碎片，为什么它有问题呢？假设第二个应用程序需要另一个分配，但由于这次所需的空间大于这两块空闲内存中的任何一块，内存管理器就不能满足需求，即使所要求的空间实际小于整个可用的空闲内存。

为了解决这个问题，Windows 内存管理器移动块（把块的内容从一个位置拷贝到另一个位置）把分散的空闲块结合成一个大的空闲块。图 4-2 说明了在图 4-1 中留下的碎片在移动前和移动后的情况。移动前，最大的可用块为 192KB。移动之后，它是 320KB。

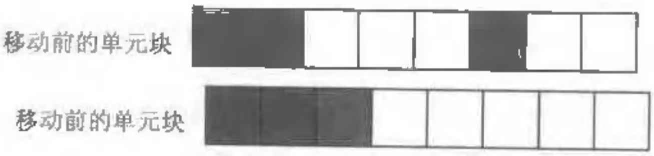

图 4-2　碎片在移动前和移动后的情况

如果所有的已分配内存都是可移动的，那么内存管理器就能有效地消除内存碎片。然而有些编程人员需要一块固定的内存，为了防止内存管理器移动这个块，大多数的固定块就会在堆中形成“沙条”（SANDBARS），并导致大量的碎片，如图 4-3 所示。固定块只能在必须需要时才使用。

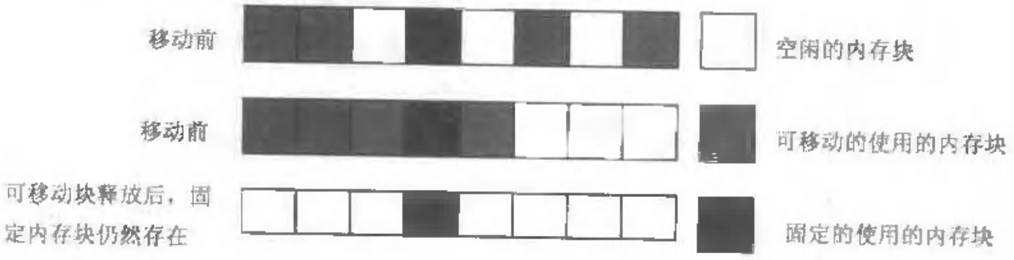

图 4-3　在堆中形成“沙条”

由于段不能直接确定物理地址，所以在保护模式下实现移动内存很容易。要移动一个段，Windows 需要把段的内容从一个线性位置拷贝到另一个线性位置，然后修改在描述符表中段的基地址。段自身的值并不变化，索引仍旧指向描述符表中的同一个入口，只有存储在描述符表中的基地址做了改变。

注意：Windows 把内存管理功能分配给两个管理部件。KERNEL 内存管理器负责线性空间使用的管理，执行前面所讲的内存压缩和丢弃。这里所讲的内存移动实际上通过 LDT（局部描述符表）或 GDT（全局描述符表）来控制。

更低层的组件——WIN386，负责管理组成物理内存。WIN386 负责与磁盘进行页的交换并负责管理工作（第 5 章和附录 A 中包括了关于 LDT 和 GDT 以及分页表的许多

信息）。这些细节对于开发人员来说并不重要，一般使用“内存管理器”来代替两个组件工作。

4.5 安全中断驱动程序内存需求

一个控制硬件中断的驱动程序对它所分配内存的类型有着严格的要求。在中断时，对应用的所有代码和数据必须是固定、页面锁定和不可丢弃的。其中包括中断处理器自己的代码，中断时所使用的在驱动程序数据段中的数据，中断时使用的动态分配的缓冲区，以及在中断时使用应用程序分配传递给驱动程序的缓冲区。

下面将进一步解释为什么中断处理器有这 3 个要求。这些需求并不只是在 Windows 下运行中断处理器才有，它们也存在于 UNIX 或 OS/2 等系统中。所以在下面几段中，将使用条款固定的“页面锁定”和“不可丢弃的”，也是其他一些操作系统所提供的一般的内存属性。

讨论了它们之后，将解释 Windows 怎样实现这 3 个属性。实现这些属性时，将使用 3 个略为不同的词：FIXED，PAGELOCKED 和 NONDISCARDABLE。这一区别是有用的，因为 FIXED 并不永远固定，而 PAGELOCKED 说明是永远固定和页面锁定。

4.5.1 固定的原因

要明白硬件中断处理器所使用的数据为什么必须在固定的段中，就需要研究下面的情况。假设中断处理器使用的段被标注为 MOVCABLE，则内存管理器可以移动它们。如果在拷贝几个字节后，发生了一个硬件中断并且中断处理器开始执行，则处理器修改在数据段（正在被移动的那个）的第一个字节中的变量，数据处理被暂停下来。当控制器完成执行后，内存管理器继续拷贝余下的部分，但是被处理器改过的变量现在已经不正确了。内存管理器把旧的值拷贝到了新段中，并且不知道处理器已修改了数值。为了避免这个问题，处理器使用的数据段必须固定。

4.5.2 不可丢弃的原因

硬件中断处理器的代码有一个不同的限制：它必须是不可丢弃的。

注意：数据段也必须是不可丢弃的。即使代码段某段为可丢弃的时，所有的数据段也不可丢弃。假设产生了一个硬件中断，内存管理器丢弃了处理器的代码段，则此时就会产生“段不存在”的错误，Windows 错误处理器就会试图再次从磁盘中调入这个段。如果当系统进入 DOS 后中断发生了，那么结果就会是试图再次进入 DOS。然而 DOS 不可能重新装入代码，所以代码段必须是不可丢弃的。

4.5.3 页面锁定的原因

硬件中段处理程序的代码和数据必须是页面锁定的，与要求段为不可丢弃的原因相似。中断处理程序访问一个可交换的（非面锁定的）的缓冲器，如果这个缓冲器已被交换到了硬盘，那么在执行中会导致一个页错误。如果中断处理程序的代码被交换到了磁盘，那么在执行中也会导致一个页错误。二者都会导致 DOS 的重新装入。

4.6 中断安全的代码和数据

中断安全的代码和数据的正确方法如下：

确保驱动程序的代码和数据段正确是实现中断安全的最简单方法，具体在驱动程序的定义文件中将代码和数据段标注为 FIXED 和 NONKISCARDABLE。注意，这一技术取决于两个众所周知的 Windows 特性。第一，Windows 在应用程序中忽略 FIXED 特性，但在 DLL 中却遵守它，驱动程序必须是 DLL（关于这一特性的详情，参见下面的"FIXED 何时是真正固定的"）。第二，当段在模块定义文件中被注为 FIXED 时，Windows 页面锁定内存以便把它固定在线性内存中。

在模块定义文件中使用关键字 FIXED 对开发者来说是简单的，但对用户来说却很麻烦。这种分配方法很容易导致下面的情况：Windows 不能开始一个新的 Windows 应用程序，并且用户得到一个错误消息：INSUFFICENT MEMORY TO START THE APPLICATION（没有足够的内存运行应用程序）。即使有足够的空闲内存和系统资源，也可能发生这种情况。

怎么会这样呢？原因是每次运行 Windows 应用程序时，Windows 加载器为一个程序段前缀（PSP）的数据结构分配一个 512 字节的块，PSP 同样也被 DOS 使用，所以它必须定位于 1MB 以下。如果 PSP 没有足够的内存可用，Windows 就不能运行应用程序。

问题是 Windows 内存管理器的对策。FIXED 块尽量从堆中的低端分配，DISCARDABLE 来自堆中的高端，MOVEABLE 在两者之间。这一策略可以帮助减少内存碎片，但经常导致 FIXED 用户用尽了珍贵的低端 DOS 内存，即使 FIXED 内存用户并不需要低端的 1MB 内存。

这里有正确的方法避免用尽低端内存。首先在模块定义文件中把代码和数据段标注为 MOVEABLE，而不是 FIXED。运行时，在中断发生前，直接固定和页面锁定段。但必须在执行第二步时小心，Windows 的 APL 函数 Global Pagelock（）会固定和页面锁定一个段。但 Global Pagelock（）会把段移动到低内存，因为 FIXED 块必须在堆中的低端，这恰好是我们想避免的。

现在所需要的就是防止内存管理器在页面锁定前把段移到低端。首先可以分配所有 1MB 以下的内存，然后调用 Global Pagelock（），再释放所有低内存。为此提供一个函数——SafePagelock（），并且将在后面仔细检验它。同时检验一个有关的问题：怎样动

态分配中断安全的缓冲区。作为驱动程序的静态段，这是一个简单和正确的方法。

4.6.1　FIXED 何时是真正固定的

在 Windows 3.0 出现以前，不论是应用程序还是 DLL 都需要 FIXED 内存——静态地在模块定义文件中使用关键字 FIXED，或动态地使用 GMEM－FIXED 标志，并获得所希望的 FIXED 内存。这一特性在 Windows 3.1 下被改变。Windows 3.1 内存管理器在应用程序中完全忽略 FIXED 特性，相反地都是 MOVEABLE 内存（然而，应用程序可以通过在分配后直接调用 GlobalFix ( )来实现 FIXED 内存）。

另一方面，当 DLL 分配 FIXED 内存时，Windows 3.1 根据实际需求真正返还 FIXED（和页面锁定）内存。应用程序和 DLL 间之所以存在不同，是由于 MOVEABLE 内存对于一个应用程序的需求来说通常是足够的。在少数需要 FIXED 内存的情况（如中断控制程序）下，功能位应该移到 DLL。

4.6.2　动态分配中断安全的缓冲区

动态分配中断安全的缓冲区的简单方法如下：

调用 GlobalAlloc ( )，并且确保缓冲区是固定、页面锁定的。GlobalAlloc ( )有两个参数：反映被分配段的属性的位映象值和段的长度。标志参数的允许值包括 GMEM-FIXED，GMEM-MOVEABLE，GMEM-DISCARABLE，GMEM-NODISCARD 和 GMEM-SHARE。

GMEM-SHARE 已在第 2 章中介绍过。虽然没有确定页面锁定属性的标志，但在 DLL 中 GMEM-FIXED 标志对页面锁定内存一直有影响，所以分配中断安全的内存应该使用标志 GMEM－FIXED/GMEM－NODISCARD/GMEM－SHARE。

注意：虽然 GlobalAlloc ( )中使用的段长度参数是一个 32 位的值，但是允许的最大分配依赖于处理机，并且远小于 2^{32}。当在 80386 以上的机器上运行时，winDos 支持的最大块长度是 16MB～64KB。

如果 GlcbalAlloc ( )返回值为 NULL，则说明这个段不能被分配，一般因为得不到那一长度的空闲块。返回的非 NULL 值是内存对象的句柄（关于句柄以及如何把它们转化为指针，将在下一节中介绍）。

这个简单方法实际上也会产生与前面关于固定的驱动程序代码和数据段类似的问题。缓冲区是固定、页面锁定和不可丢弃的，但它经常位于 1MB 以下。正确的方法是：首先在 GlobalAlloc ( )中用 GMEM-MOVEABLE 代替 GMEM-FIXED 进行分配，然后用 Safe Pagelock ( )函数固定页面锁定缓冲区，这一函数不是 WindowsAPI 中的函数。“安全”函数对于在页面锁定运行期间防止内存管理器把缓冲区移动到低内存十分必要，就像它调用 SafePagelock ( )一样。

第二部分

虚拟驱动程序简介

第 5 章　Windows 的虚拟世界

Windows 体系结构主要支持 3 种不同类型的程序：管理程序（supervisor process）、Windows 应用程序和 DOS 应用程序。为了克服这些程序之间的不相容性，Windows 在执行这些程序时（除了管理程序）在“虚拟环境中的虚拟机”上完成。在开发应用程序时，Windows 程序员可能会忽略虚拟环境和真实环境的区别，因为对于大部分应用程序来说，虚拟环境就是真实环境。

而编写 VxD 则不同，因为 VxD 是运行在管理层的，即虚拟机之外。实际上，VxD 是实现虚拟机的软件的一部分。这样 VxD 的编写者就需要更多地了解虚拟环境和真实物理环境的区别，以及 Windows 怎样创建虚拟机。还要充分了解虚拟机，对于那些需要利用一个应用程序虚拟环境中的资源来开发 VxD 的程序员来说尤为重要。

这一章将讲述 Windows 体系结构的主要特点，包括怎样实现虚拟机、虚拟环境的特征以及层环境的特征。

5.1　虚拟机是什么

虚拟资源是由系统产生的假象，它是硬件（甚至是软件）资源的仿真。为了产生虚拟资源，仿真必须是完全的，以便在编写程序时硬件是真实的，而不是仿真的。例如，虚拟内存系统使用硬盘空间、系统软件、专门的处理器性能以及少量的物理内存来仿真具有大量物理内存的系统。对于运行在虚拟环境中的程序，仿真使我们感觉到整个虚拟地址空间就像真实存在的物理内存空间一样。这样的系统叫做“虚拟化的”。

当一个系统虚拟了所有（或者几乎所有）程序可访问的资源，它就创造了一个“虚拟机”。概括起来，Windows 的虚拟机完全透明地仿真了以下的资源和性能：

- 可访问的内存空间
- 输入/输出操作
- 外围设备（显示器、键盘等）
- 中断操作

虚拟这样一个机器，而且还执行大部分代码，需要用到专门的处理器支持。80386（以及以后的兼容处理器）完成包括支持地址转换、所需内存分页、I/O 地址获取、命令获取、中断获取的完善及处理器等工作。虚拟机的管理由虚拟机管理器完成，所谓虚拟机管理器（VWW）就是主要管理程序的进程，它并不是只创建一个虚拟机，而是创建几个相互独立的虚拟机。每个虚拟机都有自己独立的虚拟环境，而且每个虚拟环境可能与物理环境根本不同。

5.2 Windows 的执行环境

Windows 在不同的执行环境中运行以下 3 种程序:

1. 管理程序
2. Windows 应用程序
3. DOS 应用程序

5.2.1 管理程序

Windows 的管理程序在第 0 层级别（最高的访问级别）的保护模式中运行，所以它们能够访问和控制实际的硬件环境。也就是说，管理程序是在实际的机器上运行，而不是在虚拟机上运行。在所有构成 Windows 的组件中，只有 VMM 和 VxD 在管理程序环境中执行。

管理程序环境是纯 32 位的。这里 32 位有两个意思：管理程序使用 32 位的寄存器（不是 AX，而是 EAX）；这些程序可以访问 2^{32}（4GB）的虚拟内存。

管理程序只使用两个选择器，每个选择器都可以访问 4GB 的虚拟内存。这两个选择器只是在属性上不同：一个被标记为 executable 并被调入到 CS 中；另一个则被标记为 non-executable 并被调入到 DS，ES 和 SS 中。内存模式的这种类型（段只被调入一次并且以后再不调入）被称为 FLAT 模式，它使得分割对程序员来说几乎不可见。

5.2.2 Windows 应用程序

第二种类型的程序（Windows 应用程序）在系统虚拟机（System VM）中运行。这些程序包括 Windows 应用程序和动态连接库(DLL)，例如 KERNEL，GDI，USER 等。Windows 部件都是普通的 DLL，所有这些程序都分享一个 SYSVM。

这些程序都运行在保护模式下，被标注为第 3 层级别(最低的访问级别)，允许 Windows“隐藏”并虚拟大部分物理资源，包括所有的外围设备和中断机制。Windows 程序具有一般的分段式内存模式（而不是 FLAT 模式），地址翻译是通过选择器和描述器来实现的。

5.2.3 DOS 应用程序

执行环境的第三种类型是 V86 虚拟机（VM86）。VM86 非常类似于 SYSVM，VM86 程序也运行在第 3 层访问级别。访问的硬件资源和中断被隐藏和虚拟，内存模式是分段的。SYSVM 和 VM86 的主要区别在于地址翻译。VM86 程序采用 8086 的翻译类型,没有选择器和描述器，段只是简单地变换成地址。

SYSVM 通常运行在保护模式，而 VM86 通常运行在 V86 模式，VM 能够交换模式。所有的 VM（包括 SYSVM）在 V86 中开始，而且所有的都会使用一个 DPMI 服务程序转

换到保护模式。在 SYSVM 中，KERNEL 总是在 Windows 初始化过程中很快地切换到保护模式。当在 Windows 下运行 DOS 扩展应用程序时，它在 V86 模式的一个 VM 中开始，然后由 DOS 扩展器转换成保护模式。

保护模式的 VM（包括 SYSVM 和一些 DOS 扩展器 VM）在访问“实模式”服务程序（如 DOS，BIOS，TSR 等）时切换到 V86 模式。

5.3 Windows 地址空间

每个 VM 都有自己的逻辑地址空间，它是与管理程序分离的。但是所有的 VM 和管理程序都分享相同的 4GB 线性地址空间。这个复杂的概念可以用一系列的图表来描述。

5.3.1 线性地址空间

在图 5-1 中给出了全部 4GB 的地址空间，而 Windows 只需要其中非常小的一部分，Windows 只使用低 2GB 中的 4MB 和高 2GB 中的一小部分地址空间。当前运行的 VM 存在于 0 到 4MB 的单元内，管理程序（VMM 和 VxD）存在于从 2GB 开始的 4MB 单元内。在这之上是所有 VM 的保护模式部分和 V86 部分。

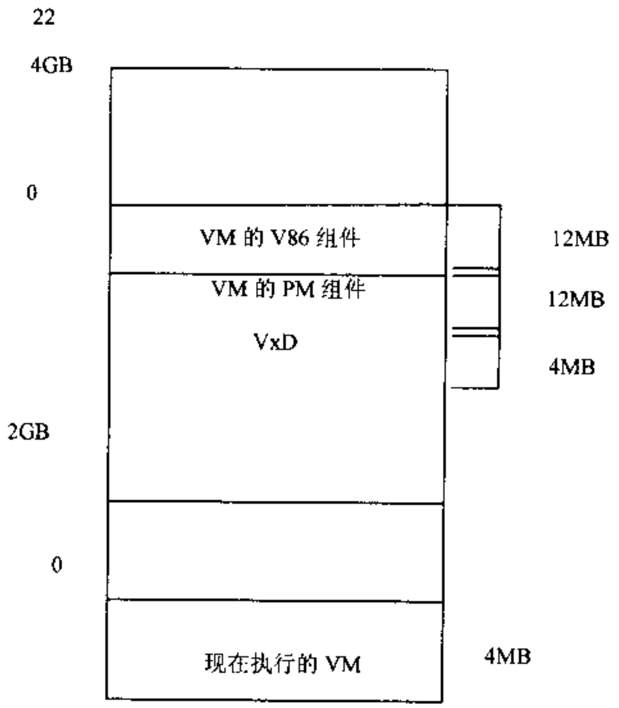

图 5-1 全部 4GB 线性地址空间描述

由于几乎整个 4GB 对 Windows 来说都不能使用，下面的图表将改变地址比例以便更

详细地说明不能使用的部分和所占用的范围。图 5-2 到图 5-4 都描述了主机中运行的 3 个 VM。VM0 是 SYSVM，它同时有保护模式和 V86 模式；VM1 是一个 DOS VM，所以它只有 V86 模式；VM2 是 DOS 扩展应用程序，所以它也同时有保护模式和 V86 模式。

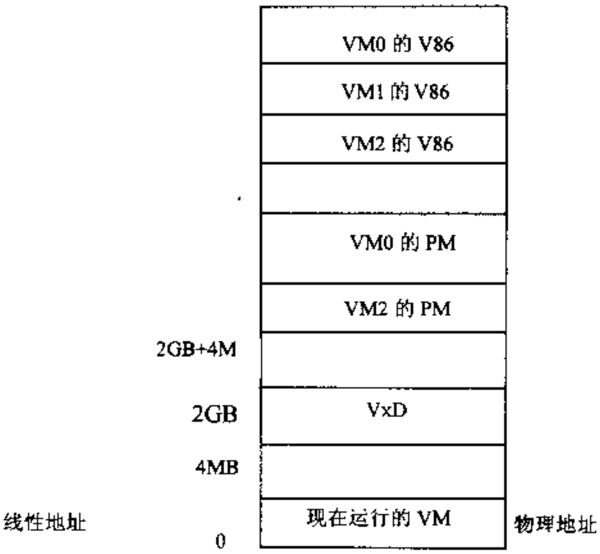

图 5-2　占用范围线性地址空间的示意图

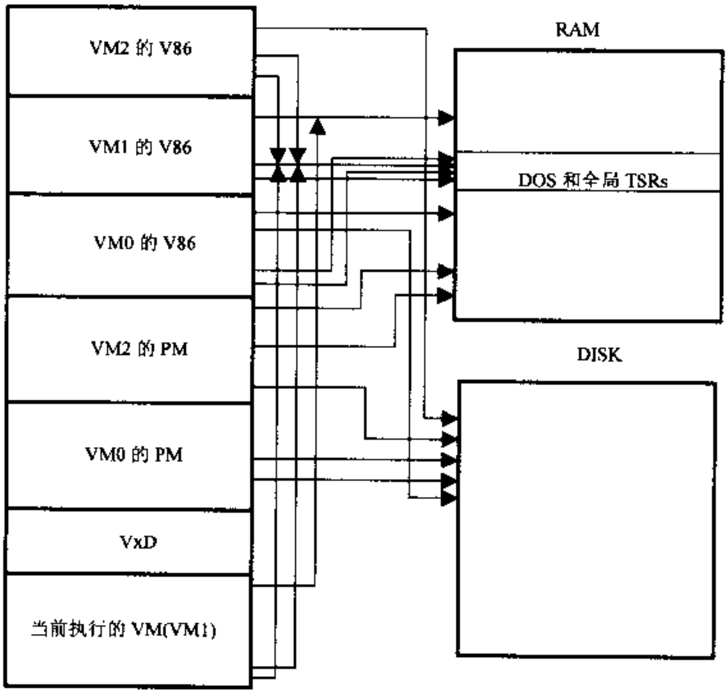

图 5-3　线性地址空间映射到物理地址空间示意图

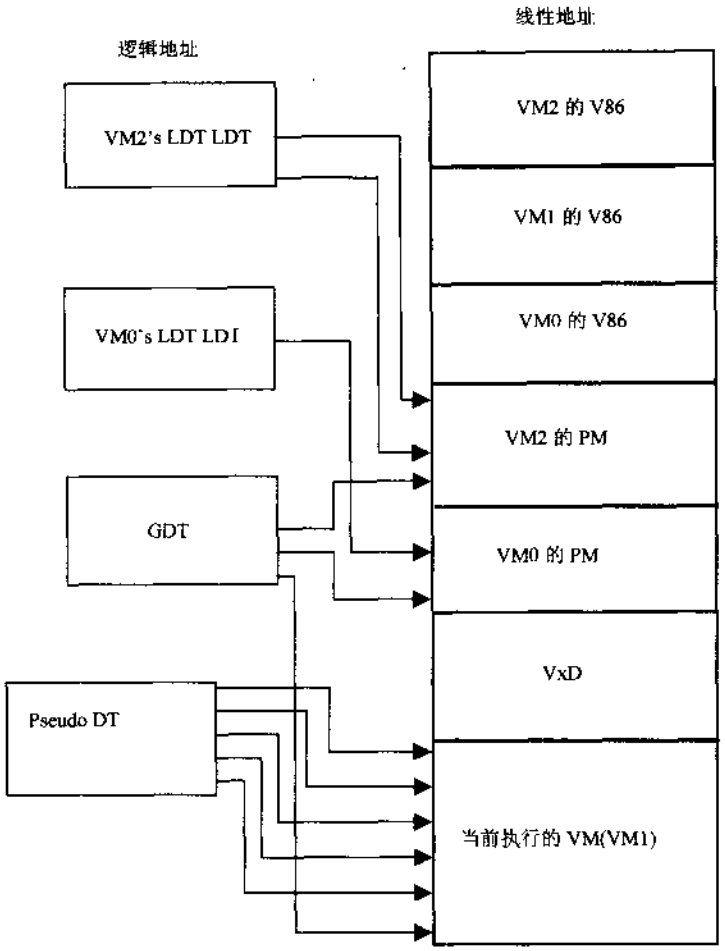

图 5-4 地址映射图

5.3.2 线性地址空间和物理地址空间

前面所讲的只是线性地址空间。线性地址空间映射到物理地址空间或者 RAM 中，并且当使用虚拟机时，也能映射到硬盘上。

图 5-3 所示的是与图 5-2 相同的线性地址空间，但是它还说明了线性地址空间怎样映射到物理地址空间和硬盘，有些 4KB 的线性单元映射到 RAM，而有些 4KB 的线性单元映射到硬盘上。Windows 应用程序和 DOS 应用程序在再次访问时可以切换到硬盘并被重新调入到 RAM 中。应该注意到，并非所有的 4KB 单元都被映射。

但是在线性空间连续的区域，在物理内存中却并不连续。实际上，线性空间在物理上很大程度是连续的，而在大部分时间里，没有人去关心物理内存是否连续。只有自由内存映射设备和 DMA 设备以及要访问这些设备的程序才关心物理地址的连续性。

图 5-3 说明了非常重要的一点：Windows 可以将线性空间中不同的区域映射到物理空间中一个相同的区域。注意，每个 V86 组件包括一个被标注为“DOS 和 全局 TSRs”的区域，它指的是在启动 Windows 之前已经存在的 DOS 下的 COMMAND.COM 和 TSR。因为每个 VM 的这个区域都映射到同一物理区域，所以它对所有的 VM 都是“全局”的。Windows 将利用这一特性保存物理内存，而不需要调入多个 DOS 的备份。

将多个现行段映射到一个物理段的能力对于 Windows 的多任务 VM 是非常重要的。物理空间的 VM1 区域在线性空间中被映射为两段：一个在 VM1 的 V86 组件内，另一个在线性空间开始的地方。

当执行某个 V86 VM 时，它在 0 到 1MB 内产生线性地址，其寻址行为定义为 V86 模式。所以如果当前执行的 V86 VM 一直使用线性地址 0 到 1MB，Windows 则必须保证线性地址 0 到 1MB 映射到正确的 VM 的代码和数据。多任务化的 DOS 应用程序的实质是采用任务开关来使页表的线性 0 到 1MB 与正确的 VM 的代码和数据相关。

每个 VM 在线性地址空间的高端区域（2GB 以上）被管理程序使用。要使用 VM 环境的 VxD，可以一直使用高端线性地址访问 VM，而不需要关心 VM 是否正在执行。

5.3.3 逻辑、线性和物理地址空间

前面讲述了在 Windows 下线性地址空间和物理地址空间的联系，不过大多数情况下，只有管理程序和处理器自身才涉及到线性地址和物理地址。Windows 和 DOS 程序通常只涉及到逻辑地址空间,通常用选择器和偏移量来描述。一个程序的逻辑地址空间就是一系列可以使用的段和选择器。VM 存在于不同的逻辑地址空间，并且映射到 4GB 线性地址空间中的不同区域。

图 5-4 给出了管理程序和 3 个 VM 使用的多个逻辑地址空间，还说明了这些逻辑地址怎样映射到线性地址，显示了 4 个描述器表：一个 GDT、两个 LDT 和一个“伪描述器表”。这些描述器表中的每个选择表都映射到线性地址中的一个区域。

就像它的名字一样，GDT 包括所有 VM 的全局描述器，而不是某特定 VM 的局部描述器。每个具有保护模式组件的 VM 都有它自己的描述器，所以每个 VM 保护模式组件的逻辑地址空间由 GDT 和 LDT 组成。

实际上，GDT 和 LDT 都是由 VMM 来维护的。V86 VM 组件不使用描述器表，但是它们的确拥有逻辑地址空间。将当前执行的 VM 设想为使用包括所有 V86 VM 产生的段的“伪描述表”非常有用。由于当前执行的 VM 的逻辑地址空间包括所有这些段，并且 V86 模式中没有实际的描述表，所以 V86 VM 在没有执行时就没有逻辑地址空间。

5.4 这意味着什么

Windows 的开发者们并不构造虚拟机，而且这个复杂的内存结构是保密的。下面给出 3 个重要的结论。

首先，VM 相互分离并与管理程序分离，一个 VM 中的程序不能访问其他 VM 的代码和数据。VM 必须使用一个选择器或者段来访问内存，一个选择器意味着一个描述表。VMM 隔离 PM（保护模式）组件以确保每个 PM 组件使用不同的描述表。V86 VM 组件不使用描述表，所以 VMM 使用不同的机制来隔离 V86 组件。V86 模式只能在 0 到 1MB 的范围内产生线性地址，所以 VMM 确保当前执行的 VM 存在于线性空间的 1MB 以下，而其他的 VM 则存在于别的地方。

其次，计时 VM 相互分离，管理程序仍然可以自由地访问它们。管理程序可以访问任何的代码和数据，因为 VxD 可以访问整个 4GB 线性地址空间的选择器。这是必须的，因为 VxD 可能会通过调整 VM 的环境来（包括 VM 的代码和数据）完成它的工作。

第三，一个全局 DOS 和 TSRVM 组件状态的改变会影响到所有 VM。

5.5 实现虚拟机

当在虚拟机中执行程序时，Windows 必须能够采集并透明地“代替”以下功能：

1. 对一个 I/O 端口查询。
2. 对一个内存映射的外围设备查询。
3. 可能导致传送到虚拟机之外的操作，如一个异常时间或中断。

为了实现所有的这些功能，Windows 特别要依赖于 80386 处理器中完善的保护模式特性。

5.5.1 捕获 I/O 操作

保护模式包括这样一些特性，操作系统可以使用捕获输入和输出指令来防止应用程序直接访问 I/O 映射的设备。在增强模式中，Windows 使用 I/O 优先级（IOPL）和 I/O 允许映射（IOPM）来控制访问 I/O 地址。

在保护模式中，每个代码段都有一个相关的描述器优先级（DPL），它存储在描述器列表中。每个代码段还有一个独立的 IOPL，当在保护模式中执行一个 I/O 指令时，处理器将段的 IOPL 和它的当前优先级（CPL）进行比较。如果 CPL≤IOPL，处理器则执行指令。否则，处理器则产生一个异常事件（一般保护错误，13h）。

在保护模式中，处理器使用 IOPM 作为保护的第二级。IOPM 是端口比特映射的列表：1 表示“拒绝访问”，0 表示“准许访问”。如果 CPL＝IOPL，I/O 将被执行，而不管端口地址；如果 CPL≤IOPL，处理器则会参考 IOPM 的值：如果 IOPM 比特被置位，将会触发一个错误；如果 IOPM 比特被清除，则执行指令。因为在 SYSVM 中程序的 CPL>IOPL，所以 IOPM 就是所有虚拟机中 I/O 的主要优先权机制。在 V86 模式中，处理器总是参照 IOPM 而不管 IOPL。

所以，通过操作 IOPM，Window 在允许自由地访问其他端口时可以捕获访问专门的端口。Windows 使用此特性来“虚拟”位于被捕获端口地址的物理设备。通过设备驱动程

序发送对设备的访问，Windows 可以保持每个使用设备的程序状态信息。

VMM 维护 IOPM。VxD 可以通过 VMM 申请捕获特殊端口的服务。VMM 通过配置 IOPM 来相应这样的申请使得 VxD 在访问端口时调用端口控制指令，并产生一个错误。如果 VxD 捕获了端口，则几乎可能发生任何事情：VxD 可能忽略指令，可能执行指令，可能替换值（OUT 3F8，01 可能变为 OUT 3F8，81）。

Windows 和它的组件可以捕获几乎所有标准的 PC 的 I/O 设备，而不能捕获非标准的 I/O 设备。表 5-1 列出了可捕获的端口位置。一个第三方的 VxD 可以捕获其他端口。

表 5-1　增强模式 Windows 模式的 I/O 位置

端口位置	设　备	获　取
0h－1Fh	DMA 控制器 1	VDMAD
C0h－DFh	DMA 控制器 2	VDMAD
80h－8Fh	DMA 页寄存器	VDMAD
20h－21h	主 PIC	VPICD
A0h－A1h	从 PIC	VPICD
40h－44h	定时器	VTD
60h，64h	键盘	VKD
3F8h－3FFh	串口 COM1	VCD
3C0h－3DFh	VGA	VDD

5.5.2　捕获内存操作

80386 的优先级和段的机制在每个访问内存的命令执行期间连续执行几个检查操作。尽管如此，Windows 还是主要依靠页错误机制来访问虚拟内存。为获取对当前内存位置的访问，Windows 会在页表中包括内存位置的页入口处将页标注为“不可用的”。当在虚拟环境中运行的程序试图访问这个页时，就会产生一个错误。错误会被一个可以检查错误命令的控制器捕获，并且决定产生与虚拟环境的请求相关联的动作。

Windows 使用这一机制来虚拟如视频缓冲区这样的虚拟内存映射设备。当 DOS 的程序写位于逻辑地址 B000:0000H 的视频缓冲区时，输出并不显示在屏幕上，因为 Windows 获取了写，并且将其放入物理内存的另外位置。

5.5.3　捕获中断和异常事件

为了获取内存和 I/O 访问，Windows 需要获取正确的有优先级的指令。这些指令可能会旁路处理器的安全特性或者干扰虚拟机的完整性。优先级指令包括影响处理器中断标志（CLI，STI，POPF，IRET）或软件中断的指令（INT n），以及那些调入描述器表的指令（LLDT，LGDT，LIDT）。Windows 获取这些指令以保证 VM 的完整性。对应于 *INT* 指令，Windows 开发了透明地截取 DOS 和 BIOS 调用的软中断。

在虚拟机中的程序运行在第 3 层中。在这里执行一个有优先级的指令，就会产生一个异常事件。这时处理器会切换到第 0 层，并且将控制交给相应的控制器。

每个段都有一个相应的描述器表优先权等级（DPL）。段的优先权等级决定大部分指令的优先权等级（例如 LLDT，LGDT）。另外一些指令则从 IOPL 中得到它们的优先权等级，例如在第 3 层中的程序执行 STI 或 CLI 指令，只有 CPL>IOPL 时，处理器才会触发一个异常事件。

VM86 环境和 SYSVM 环境的主要区别之一就是涉及这些基于 I/O 的优先权。当 386 体系结构支持在两种情况下捕获 CLI 和 STI 指令时，Windows 不能在 VM86 模式下捕获 CLI 和 STI 指令。WIN386 为 DOS 应用程序设置为 CPL＝IOPL，所以 CLI 和 STI 不能产生异常事件。有趣的是，虽然 Windows 以 CPL>IOPL 条件运行 Windows 应用程序，VMM 仍然可以正确转换并再次允许中断。

5.6 V86 模式简介

DOS 应用程序对于“接管整个 PC 机”是不能胜任的。几乎每个 DOS 程序都使用所有可用的系统内存,并且直接对视频内存进行写操作，而不是系统调用。DOS 程序经常直接影响端口映射的硬件设备，例如规划 PC 机的中断控制器或硬件定时器。

VMM 必须能够防止 V86 程序直接访问重要的设备。几个不同的应用程序可能都需要使用音频适配器，但每个又有一些不同的配置。一个可靠的硬件定时器位于多任务内核的中心。VMM 使用上述的特性来维持对难以驾驭的 DOS 应用程序的适当控制。

通过维持每个虚拟机不同的页表，VMM 将不同 VM 的地址空间映射到物理内存的不同区域。VMM 还使用此特性来重定向像视频缓冲区这样的内存映射设备。当 DOS 的程序写位于逻辑地址 B000:0000H 的视频缓冲区时，输出并不显示在屏幕上，因为页表将内存访问重新隐射到物理内存的另外的位置。

VMM 提供一个 8086 兼容的地址空间以便可以将处理器切换到 V86 模式，在这里线性地址可以通过移位-相加的算法计算出来，如图 5-5 所示。

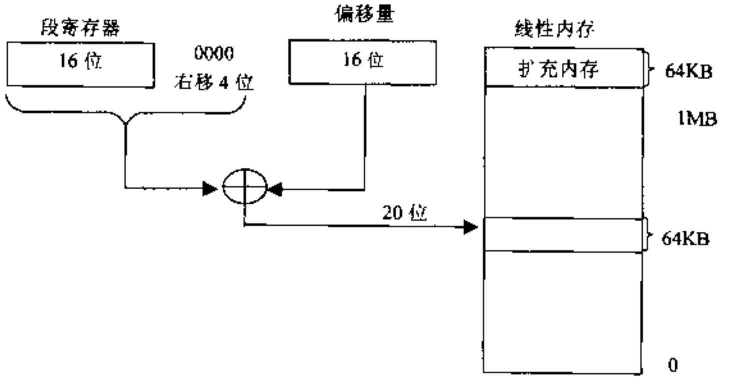

图 5-5 V86 模式地址转换

VMM 通过 I/O 允许隐射（IOPM）来防止 V86 访问重要的 I/O 端口，包括程序“禁止进入的”所有端口的列表。如果 V86 程序访问其中的某个端口，处理器就产生一个异常事件。

总之，VMM 通过捕获软件中断指令来截取 DOS 和 BIOS 的调用。

5.7 总　结

VM 是 Windows 结构体系的中心。不论是 DOS 的兼容性还是 Windows 的多任务能力，都依赖虚拟机产生的虚拟函数和过程隔离。Windows 依靠 80386 的各种硬件能力（特别是保护机制）来实现它的虚拟机，但是开发人员必须注意虚拟机逻辑能力和 80386 物理能力的区别。例如，Windows 的 VM86 并不是简单地在 386 的 V86 模式下运行，就像 SYSVM 不是在 386 的保护模式下运行一样。

因为 VxD 在虚拟机之外运行，所以它们不受关联应用程序代码的保护机制的约束。这个自由性意味着更大的能力和责任。当 VxD 可以“到任意地方做任何事”时，它必须非常仔细地做，以免破坏操作系统。要正确实现这个优先的角色，VxD 必须遵守专门的结构要求。第 6 章将介绍这些要求以及生成 VxD 的特殊工具。

Windows 一些重要的扩展模式特性是 VM 扩展模式执行的直接产物。

第 6 章　虚拟驱动程序简介

VxD 是一种特殊的只能在扩展模式中使用的设备驱动程序。与第一部分介绍的基于 DLL 的驱动程序不同，VxD 与虚拟机管理器（VMM）一起安装，一起合作，一起分享执行环境的一些组件。

由于与操作系统的这一特殊联系增加了在 Windows 和 DOS 应用程序中没有的能力。VxD 可以不受限制地访问所有硬件设备，可以自由地检查操作系统的数据结构（如描述器和页表），并且可以访问一些内存地址。VxD 还可以捕获软件中断，捕获对 I/O 端口和内存区域的访问，甚至可以截取硬件中断。

要支持这一核心集成的级别，所有的 VxD 必须具备以下条件：

- 符合标准的结构
- 和 VMM 一起注册它们的服务
- 服务至少是专门的消息协议的一部分

本章将讲解 VxD 的这些基本特性，以后各章介绍 VxD 怎样实现不同设备的相关能力。

6.1　VxD 的基本结构

虽然 VxD 使用 32 位的内存模式，但是它的程序和数据仍然可以被组织成段。VxD 使用段的这些类型：

- 实模式初始化
- 保护模式初始化
- 不可分页的
- 可分页的

除了实模式初始化以外，每个段类型都有一个代码段和一个数据段，VxD 可以有所有的 7 个段。实模式初始化是 16 位的，其他的段是 32 位的。

WIN386 加载器期望以一个专门的顺序发现这些段，还期望发现一个重要的数据结构，即在“不可分页的”段中的特殊偏移量的设备描述器部件（DDB）。

VMM 在切换到保护模式前，在 VxD 的“实模式初始化”段的开始处执行程序。这给了 VxD 一个机会可以监测 Windows 的实模式环境并且决定 VxD 是否继续加载。通过在 AX 中返回的退出代码，VxD 可以告诉 VMM 是继续加载 VxD 的保护模式部分还是退出加载 VxD，或者甚至退出加载 Windows.

在每个 VxD 的实模式部分执行后，VMM 切换到保护模式，并且给 VxD 一个机会使它在自己的“保护模式初始化”段中执行程序。保护模式初始化程序也能返回一个错误代码，以通知 VMM VxD 初始化失败。如果 VxD 报告初始化失败，VMM 就将 VxD 标注为

无效并且永远不再调用它。

不论是“实模式初始化”段还是“保护模式初始化”段，在初始化完成之后都是可丢弃的。这些段在第一个 VxD 初始化前被加载，并且在所有的 VxD 初始化完成之前不能丢弃。

大多数的 VxD 位于其他两个段中的一个（“不可分页的”和“可分页的”），而且它们在初始化完成之后不能丢弃。就像它的名字一样，“可分页的”段可以被虚拟内存管理器分页到硬盘，“不可分页的”段则被永远不能分页到硬盘。

实际上，Windows 3.x 认为可分页和不可分页的 VxD 的段是一样的：它永远不会将任何段调到硬盘。尽管如此，Microsoft 希望在 VxD 中区分这两种类型。Windows 3.1 的设备驱动程序开发包（DDK）特别声明 VxD 有“不可分页的”段和“可分页的”段，并且清楚说明以下必须是“不可分页的”段：

- 设备控制程序
- 设备描述表部件
- 硬件中断处理器和被它们访问的数据
- 可能被硬件中断处理器调用的服务，例如“异步”服务

虽然 Windows 3.x 不将可分页的 VxD 段调入到硬盘，但如果人们按照要求编写 VxD：Windows 95 就能将可分页的 VxD 段调入到硬盘。所以当在 Windows 95 中装入 VxD 时，它会马上正确运行，并且减少许多麻烦。

6.2　设备描述器部件

设备描述器部件（DDB）是 VMM 给 VxD 的“句柄”。它一直包括确定 VxD 和指向 VxD 标准入口指针的信息，还可能包括指向一个或更多的被应用程序，或其他 VxD 使用的非标准入口指针。DDB 必须通过宏 Declare_virtual_Device 来静态地初始化。由于 DDB 只使用不可分页段中的同一个偏移量，所以 DDB 和它的入口在 VxD 加载后就能被 VMM 使用。表 6-1 给出了所有 DDB 结构的字段。

表 6-1　DDB 结构的字段

字 段	说　明
名称	8 个字节的 VxD 的名字
主版本号	VxD 的版本，与 Windows 版本无关
子版本号	VxD 的版本，与 Windows 版本无关
设备控制程序	设备控制程序的地址
设备 ID 号	与被代替的 VxD 的 ID 相同，或者是由 Microsoft 确定的唯一值
初始化顺序	通常使用 Undefined_Init_Order。为强迫在一个具体的 VxD 之前/之后初始化，在 VMM.INC 中指定 Init_Order,并且加/减 1
f 服务表	服务表的地址
V86 API 程序	V86 API 程序的地址
PM API 程序	PM API 程序的地址

设备控制程序（DCP）字段包括进入 VxD 的主要入口地址。在实模式初始化之后，所有从 VMM 发出的到 VxD 的调用都通过这个入口。VMM 使用这个入口通知 VxD 在 VM 和 Windows 中状态的改变，而 VxD 则通过反应这些事件来完成它们的工作。

VMM 用 DDB 设备 ID 号字段来识别 VxD。选择设备 ID 的规则如下：如果 VxD 是直接替代一个已经存在的 VxD，则采用 VMM.INC 中已存在的 VxD 的 ID 号；如果 VxD 是一个新的 VxD，则需要向 Microsoft 申请一个自己的 ID 号；如果一个 VxD 输出一些入口到应用程序或者其他的 VxD，则必须获得一个自己的独一无二的 ID 号。VMM 使用这个 ID 号可以正确决定 API 的入口。另一方面，如果 VxD 没有输出入口，可以只在 VMM.INC 中使用 Un_Defined_Device_ID。

如果一个 VxD 为应用程序提供了一个 API，那么 DDB 就包括了 API 入口的地址，还可以提供保护模式的 API（为 Windows 应用程序）、V86 模式的 API（为 DOS 应用程序）。DDB 为每种 API 提供一个字段。由于每种类型应用程序的 API 入口只有一个，VxD 专门在寄存器中使用一个函数代码来决定调用者所需的特殊功能（如 DOS 下的软件中断）。

VxD 可以输出不同的入口，以提供给其他的 VxD 使用。DDK 文件把它称作“服务”，而不是 API。这个服务与 DDB 包含的服务表中的 API 不同，不是一个简单的服务入口。服务表主要是函数代码和函数地址的列表。VMM.INC 中包括可以方便地产生这个表的宏 Begin_Service_Table 和 End_Service_Table。

6.3　支持数据结构

DDB 实际上只是 Windows 需要的关于 VxD 的一个数据结构。由于 VxD 为多个设备提供特殊的服务，并且与多个 VM 相互作用，所以大多数 VxD 都创建自己支持的数据结构，以存储每个设备和 VM 的配置以及状态信息。

和第一部分讲的 DLL 驱动程序一样，VxD 专门使用一个或更多的上下文结构存储 I/O 基地址、IRQ 等。VxD 的设备上下文结构可以在 VxD 的数据段中静态地分配，或者通过 VMM 服务动态地分配。

实际上，如果设备的数量一直固定不变，则静态地分配设备结构；如果设备数量是变化的，则动态地分配设备结构。例如所有的 PC 都有两个 DMA 控制器，所以 VDMAD 在它的数据段中静态注明设备结构；但串口的数目可以从 1 到 4，所以 VCD 为它找到的串口动态地分配。

如果你想在运行期间动态地分配设备结构，可以使用 VMM 服务 HeapAllocate。然而，如果设备结构包括一个大的缓冲区（4KB 或更大），你应该在设备结构只包括一个指向缓冲区的指针，并且使用 PageAllocate 单独分配大缓冲区。HeapAllocate 分配小的缓冲区，PageAllocate 分配大缓冲区。这里的“小”和“大”是针对 4KB 而言的。

虽然对每个设备信息的管理是一个相似的概念，但管理每个 VM 或者每个设备/VM 的信息非常困难。幸运的是，VxD 可以申请 VMM 代替管理每个 VM 的存储。对于任意一个 VM，VMM 能够分配和使用控制块（CB），VxD 可以使用 VMM 服务在 VM 的 CB

中存储自己的每个 VM 数据区。

VxD 调用 VMM 的服务 Allc_Device_CB_Data 并申请一定大小的块来保存这个 CB 空间，VMM 将在整个 CB 中返回分配块的偏移量。一旦 VxD 申请这一空间，VM 就将在每个 VM 的 CB 中的相同偏移量地方存储它。由于 VxD 将一直访问当前 VM 的句柄（在 VM 的 CB 的起始处），所以它能够一直定位这个 CB 的数据。图 6-1 给出了控制块数据（CBD），用来存储每个 VM 的状态信息。

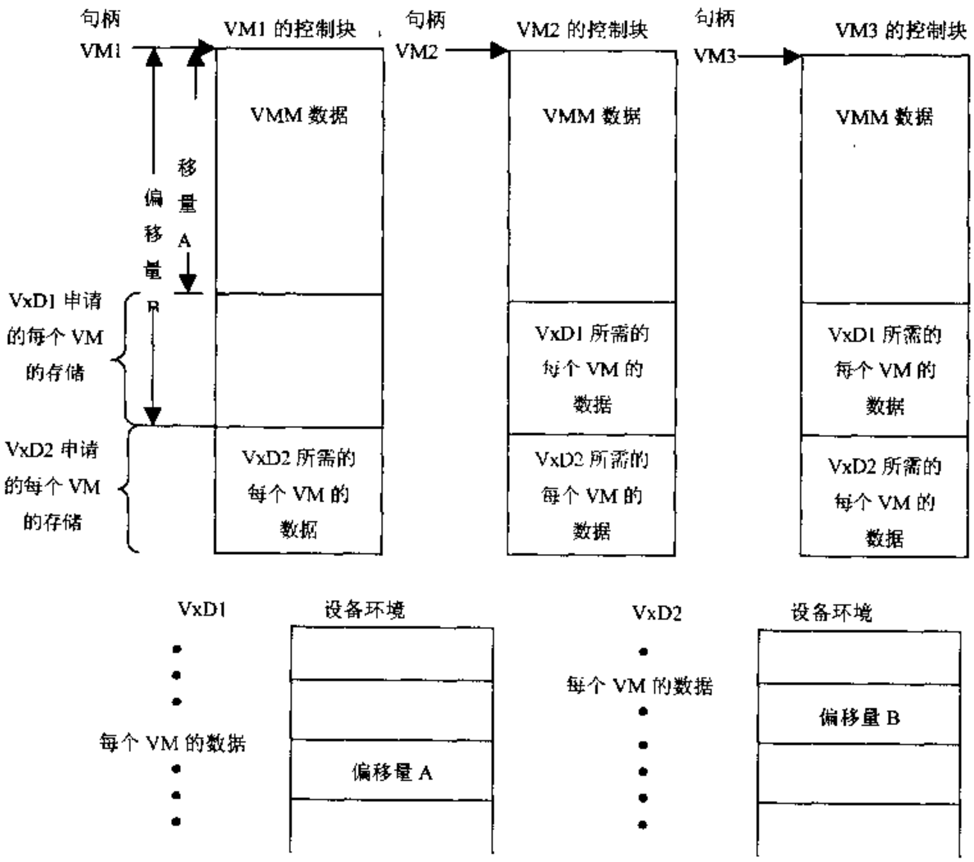

图 6-1　控制块数据（CBD）存储每个 VM 的状态信息的示意图

另一个重要的被 VxD 使用的数据结构是客户寄存器结构（CRS），这个结构包括 VM 的当前寄存器和标识。如果要检查或修改一个 VM 的状态，VxD 需要检查或修改 CRS 中的寄存器。如果 VxD 提供一个 PM 或 V86 模式应用程序使用的 API，那么它对 VM 的状态特别感兴趣。VxD 通过这些客户寄存器来得到它的输入和提供它的输出。由于 VMM 在调用 VxD 的 API 入口前将 EBP 设置为指向 CRS，所以大多数的访问是通过 EBP 来实现的。VxD 也可以在 VM 的控制块中通过用 CB_Client_Pointer 发现的地址来发现 CRS。图 6-2 给出了它们的关系。

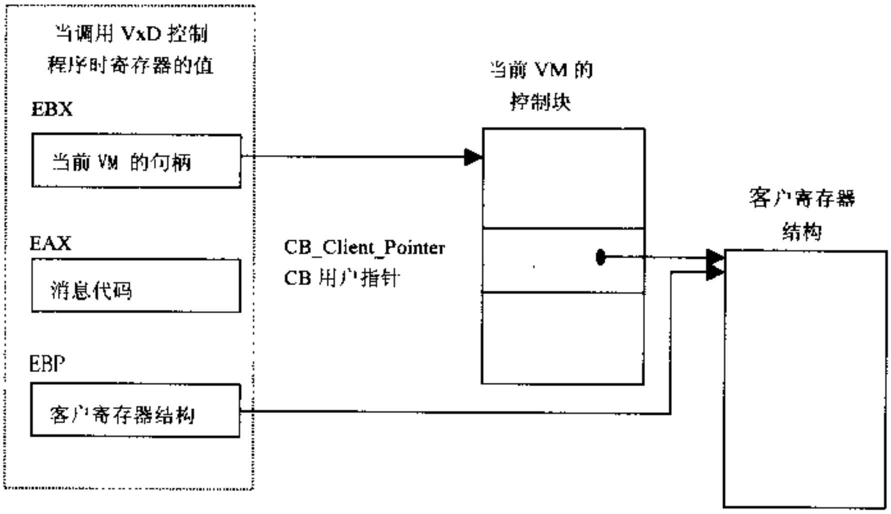

图 6-2 当前 VM 句柄、VM 的控制块和客户寄存器结构关系示意图

6.4 事件的标志信息

在实模式初始化完成之后，需要通过一个专门的接口通知 VxD 有关的事件。为了向 VxD 发送消息，VMM 存储在 DDB 中的指针调用 VxD 的 DCP 将消息代码送到 EAX 中，将当前 VM 的句柄送到 EBX 中。宏 Begin_Control_Dispatch 和 Control_Dispatch 以及 End_Control_Dispatch 可以很容易地建立一个转移表来完成这些事情。VMM 使用这一接口来报告初始化的 VxD、清除活动以及 VM 状态改变。

大概有 5 类共二十多个消息。表 6-2 给出了这些消息和它们所属的类型。在两种初始化类型和两种结束类型中，消息一直按照所列的顺序发送。这些消息的寄存器参数以及返回值定义可以在 Windows 的 DDK 文件中找到。

表 6-2 VMM 通知相关事件 VxD 的消息分类

类 型	消 息	说 明
系统初始化	Sys_Critical_Init	关中断并保持；做最小处理
	Device_Init	SYSVM 已创建，VxD 做大部分初始化
	Init_Complete	在 VxD 执行 Device_Init 后所需的一些处理
系统终止	System_Exit	SYSVM 被取消，但仍在内存中
	Sys_Critical_Exit	SYSVM 不再在内存中，关中断
VM 初始化	Create_VM	VxD 初始化每个 VM 的数据
	VM_Critical_Init	关中断
	VM_Init	VM 完全建立，VxD 可在 VM 中调用实模式程序
	Sys_VM_Init	与 VM_Init 相同，但 VM 是 SYSVM

续表

类 型	消 息	说 明
VM 终止	Query_Destroy	VM 异常终止；如果 VM 没有取消就返回标识 Carry
	VM_Terminate	VM 正常终止；VM 仍然存在，所以 VxD 可在 VM 中调用实模式程序
	Sys_VW_Terminate	与 VM_Terminate 相同，但 VM 是 SYSVM
	VM_Not_Executable	异常和正常终止，VM 仍在内存中，但不执行
	Destroy_VM	VM 不再在内存终止
VM 状态	VM_Suspend	VM 被另一个 VxD 暂停；VxD 应该放弃
	VM_Resume	VM 被恢复
	VM_Set_Device_Focus	VM 有键盘/鼠标响应
	Begin_PM_App	VM 已经开始一个保护模式（DPMI）应用程序
更换	End_PM_App	VM 已经结束一个保护模式（DPMI）应用程序
其他	Reboot_Processor	只被虚拟键盘驱动程序控制
	Debug_Query	表示产生一个调试器，VxD 显示状态

许多 VxD 只是控制这些消息。第 7 章将讲述 VxD 怎样控制这些消息。大部分消息在 VxD 或 VM 的使用期内标记重要的事件，下一节将讲述这些消息怎样与它服务的 VxD 和 VM 的使用周期相联系。

6.5 保护模式初始化

在 Windows 初始化期间，VxD 将收到 3 个消息，每个消息标注 3 个初始化阶段中的一个。这 3 个消息的响应与这些阶段紧密相关，VxD 通过返回 Carry 值来表示失败。此时 Windows 不会调入 VxD，VxD 也不会收到其他消息。

Windows 初始化的第一个阶段是被 Sys_Critical_Init 消息标注。这时会发生几种活动。中断被关闭，所以如果设备需要在中断关闭时初始化，那么就在这个阶段执行。如果密度 VxD 为其他 VxD 提供服务，那么就对这个消息在控制器中做实现这些服务所需的初始化，因为其他 VxD 可能会在接到这个消息后立即调用服务。VxD 还应该从 SYSTEM.INI 文件中检索配置信息、环境参数和专用文件，因为 VMM 的初始化信息服务只在这一阶段有效。如果设备需要在 VMM 内存的特定范围进行操作，就在这一阶段申请内存。必须将其他一些活动延迟到下一个阶段，而且 VMM 的一些在 VM 中执行程序的服务（如 SimulateInt 或 ExecInt）在这个阶段无效，因为 VM 还没有创建。

下一个消息 Device_Init 在 VMM 创建 SYSVM 后产生，它通知第二个初始化阶段的 VxD。大多数的 VxD 的启动在这个阶段执行。在这一阶段，VxD 要分配设备结构和 CB 内存，挂钩端口可中断。由于 SYSVM 已经建立，所以可以在 SYSVM 中使用 SimulateInt 和 ExecInt 执行软件中断。

Init_Complete 标注系统初始化的最后一个阶段，通常只有需要在 VM 内存中分配或

申请页的 VxD 才响应这个消息。

另一组消息跟踪 VM 的运行期。一个新的 VM 的创建也需要 3 个阶段，每个阶段都有自己的消息：Create_VM，VM_Critical_Init 和 VM_Init。

当 VxD 接收到第一个消息 Create_VM 后，它将初始化一些与 VM 相关的数据。VM_Critical_Init 标注下一个阶段，到消息 VM_Critical_Init 的一个错误响应将导致一个 VM 结束序列消息（VM_Not_Executable 消息）。如果 VM_Create 失败，则没有 VM 结束序列消息产生。最后一个阶段是 VM_Init。这时 VM 已经建立，而且 SimulateInt 和 ExecInt 也可以在新创建的 VM 中使用，以调用软件中断。

一个 VM 的取消也需要 3 个阶段。声明 VM“死亡”的消息 VM_Terminate 导致一个 VM 顺利地退出。当 VM 仍然存在时，VxD 将通过 SimulateInt 和 ExecInt 做一些动作。下一阶段 VM_Not_Executable 在正常退出和异常退出时都可以发生。EDX 寄存器中包含说明结束原因的标识值。这些标识制在表 6-3 中列出。由于 VM 已经结束，所以 SimulateInt 和 ExecInt 不能使用。Destroy_VM 标记最后一个阶段。如果 VxD 不关心 VM 结束的原因，并且不需要 SimulateInt 和 ExecInt，VxD 只需要响应最后的消息。

表 6-3　在 EDX 寄存器中包含的指示终止的标识值

VNE_Crashed	VM 崩溃
VNE_Nuked	VM 在仍然活动使用时取消
VNE_CreateFail	VxD 在 Create_VM 时失败
VNE_CrInitFail	VxD 在 VM_Critical_Init 时失败
VNE_InitFail	VxD 在 VM_Init 时失败
VNE_Closed	VM 完全关闭并取消

在 SHELL 异常关闭一个 VM（特别是响应用户的申请）前，它将发出消息 Query Destroy。VxD 可以通过 Carry 值响应这个消息来声明 SHELL 不能取消 VM。这时 VxD 还要通过使用 SHELL 消息服务通知产生问题的用户。

除了 VM 的建立和结束事件外，VxD 还通知改变当前运行的 VM 的调度事件。当 VMM 的调度程序暂停和恢复一个 VM 的运行时，消息 VM_Suspend 和 VM_Resume 发送到 VxD。

6.6　创建 VxD 的工具

第一部分中讲的 DLL 驱动程序只是 DLL，采用标准的 Windows 开发工具可以很容易地用 C 语言结合汇编程序写成。而编写 VxD 则需要不同的工具。创建 VxD 需要 32 位的编译器和汇编器、32 位的连接器以及一个可以通过增加一个特殊的首部将 32 位的.EXE 文件转换成 VxD 的 ADDHDR 工具。这些工具在标准的 Windows 开发工具包中（如旧的 Windows SDK 或新的 Visual C++）并没有全部提供。

当 Windows 出现时，所有的 VxD 都用汇编语言写，因为当时没有 32 位的编译器。现在有了 32 位的 C 编译器，但是用 C 编写 VxD 需要一些技巧，因为 VxD 不是一般的 32

位 Windows 应用程序。然而，一些第三方的工具包可以用来完成这些技巧。

如果用汇编语言来编写 VxD，可以有几种选择。Windows DDK 包括所有用汇编语言编写 VxD 所需要的东西：32 位的汇编语言、32 位的连接器和 ADDLDR 工具。如果你要使用 32 位的汇编语言，那么它会带有 32 位的连接器，但是没有 ADDLDR 工具。ADDLDR 工具只能从 Microsoft 的 DDK 中获得。

由于第一部分的例子全部是用 C 编写的，所以你可能会对全部用汇编语言编写的 VxD 感到惊讶。许多 VxD 可以用很少几行相关的代码实现，汇编语言的 VxD 例子很好理解。

如果还需要一个查看 VxD 工作的调试器，而且使用 CodeView 不行，可以有两个选择：DDK 中的 WDEB386 调试器或者是 SoftIce for Windows。

6.7 总　　结

尽管在 VMM 和 VxD 服务索引中列出了上百个函数，但是对于许多 VxD 来说，不需要知道在本章中提到的以外的东西，除非设计者还要做一些非常特别的事情。

第 7 章　设备选优 VxD

最经常、最直接使用 VxD 的情况之一是在多个 VM 中选优设备的使用。多任务时，有可能两个程序试图同时使用一个设备。如果每个过程都必须通过同一个 VxD 访问设备，那么驱动程序可以将访问串行化。

不幸的是，一些应用程序（特别是 DOS 应用程序）不是通过调用操作系统的驱动程序，而是直接对硬件进行操作。虽然如此，因为 VxD 依靠机器的端口访问和页访问硬件并且不是显式调用设备驱动程序，VxD 可以中断 VM 对设备的访问，甚至是 DOS 应用程序的直接操作。这样，VxD 可以可靠地检测到多个 VM 试图同时访问一个设备的情况。当然，必须注意到 Windows 并不依靠 VxD 来检测多个 Windwos 应用程序试图访问一个设备的冲突，因为端口访问和页访问特性工作在每个 VM 的基础上，而且 Windows 应用程序都位于同一个 VM 中。把多个 Windows 应用程序对一个设备的访问串行化是 Windows 驱动程序 DLL 的工作。

一旦冲突被检测到，VxD 就会采取相应的行动。大多数情况下，VxD 仅仅是警告争用的用户，并且询问用户选择哪个应用程序来访问设备。

本章使用两个 VxD 的例子来说明怎样对端口映射和内存映射的设备争用进行判优。第一个例子 PORTTRAP 说明怎样建立对 I/O 端口的占用，第二个例子 PAGETRAP 说明怎样建立对内存页的占用。

7.1　VMM 和处理器的异常事件

在 Windows 开始时，VMM 在 IDT 中为所有的处理器异常（包括错误、争用和中断）安装处理器。VxD 可以使用一系列的 VMM 服务来进行注册，注册内容的来源是当一个特别错误、争用或中断发生时，从 VMM 来的通知。实际上，不是 VMM 而是生成另一个 VxD。虚拟可编程的中断处理器设备（VPICD）提供注册硬件中断的服务。接着 VMM 会在异常事件的初始期进行处理，然后调用已注册的 VxD。

VMM 提供一个通用的可以挂钩任何类型的错误、争用或者中断的服务 Hook_VMM_Fault_Handler()。VMM 还对于端口占用处理器、页错误处理器和硬件中断处理器专门有几个入口点。通过使用这些特殊的服务，VxD 可以利用由 VMM 错误处理器执行的前期工作，来找出哪个 VM 导致异常事件，哪个端口或页被 VM 访问，以及甚至是导致争用/错误的指令。

本章的 VxD 使用 Install_IO_Handler（）和 Hook_V86_Page（）两个不同的服务来接收 I/O 端口访问和内存页访问的回送值。第 8 章我们将使用可以使 VxD 获得硬件中断回送的 VPICD 服务 VPICD_Virtualize_IRQ（）。

7.2 端口所有权

两个 VxD 对设备管理的使用的规则非常简单，简单地说就是“接触到它，就占有它，直到消失。”这就意味着第一个访问设备的 VM 就被声明为占有者 VM，而且其所有权直到 VM 结束才释放。如果其他的 VM 试图访问它占有的设备时，VxD 就会询问用户选择哪个 VM 是所有者（即占有者）。

设备所有权的概念在 VxD 中非常普遍。VxD 通过禁止本地端口 I/O 的获取来允许所有者 VM 直接访问设备，如果设备执行中断，VxD 确认只有所有者 VM 能看到它们。

7.2.1 PORTTRAP（端口访问）程序例子

通过编写 VxD 可以说明端口争用实际上很复杂，因为 Windows 包括的 VxD 虚拟化了大部分标准 PC 的 I/O 端口设备，而且 VMM 只允许一个 VxD 占有给定端口的访问。为了不取代已存在的设备，本章的 PORTTRAP 占用地址为 400H 的假想设备 I/O 端口，在扇入了 3 个 I/O 地址。

在本章的两个例子中，PORTTRAP 讲述的更加详细。它在 VMM 的控制块中分配每个 VM 的存储，并且允许用户在 VM 之间决定争用。程序例 7.1 和例 7.2 所列的 PORTTRAP 需要非常多的代码。

PORTTRAP 的 VxD 由 4 个组件组成：消息分配器、Device_Init 消息的控制器、VM_Not_Executable 消息的控制器以及占用控制器。由于 VxD 只响应消息 Device_Init 和 VM_Not_Executable，所以消息分配器组件是无足轻重的。初始化消息控制器和相关占用控制器是最不可缺少的。下面对程序进行详细讲解。

1．初始化例程

PORTTRAP 的 Device_Init 命令调用 VMM 的服务 Install_IO_Handler（）来为每个占用的端口注册端口占用控制器。一旦注册，只要 VM 访问被占用的端口之一，VxD 的占用控制器就被调用。图 7-1 给出了调用 Install_IO_Handler（）的寄存器参数。

```
Install_IO_Handler
输入：
EDX＝被捕获的端口
ESI＝当 VM 访问端口时调用的程序的地址
输出：
Flags＝Carry 设置（如果端口已经被挂钩或者脱离了挂钩）
```

图 7-1 调用 Install_IO_Handler 的寄存器参数

PORTTRAP 在一个循环中调用 Install_IO_Handler（），每次调用在 ESI 中传递同一个指向命令 Port_Trap_Handler 的地址指针，在 EDX 传递不同的端口数。

捕获多个端口的方法是使用 VMM 的服务 Iinstall_Mult_IO_Handlers（），这个服务

在 EDI 中取一个指向包括端口号以及与其相关的回调程序的表参数。VMM.INC 中的宏 Begin_VxD_IO_Table 使定义端口和回调的定义微不足道。

因为 Port_Trap 对几个端口只使用一个处理器，当它被调用时需要在进行适当的活动前决定访问哪个端口。Port_Trap 通过给每个端口一个自己的回调例程（通过 Iinstall_Mult_IO_Handlers( )或在每个 Install_IO_Handler( )的调用中传送不同的例程）。

Install_IO_Handler( )最初能够捕获所有的 VM，包括现在和将来的。稍后 VxD 可以用 VMM 的服务 Disable_Global_Trapping（）和 Enable_Global_Trapping（）改变全局的捕获状态。这两个服务的唯一参数是 EDX 中的端口号，使用它们可以禁止或者允许对某个特定 VM 的捕获，用 EDX 中的端口号和 EBX 中的 VM 控制。

Device_Init 的最后一步是使用 VMM 的服务 Alloc_Device_CB_Data（）在控制块中分配每个 VM 的存储，接着 Device_Init 将分配的控制块的偏移量存到变量 CB_Offset 中。PORTTRAP 对于每个 VM 数据只需要一个 DWORD，这个 DWORD 包括一个标志字段。

2．争用处理器

争用处理器在捕获阶段总是通过 VMM 的回调被直接调用。由于端口操作可以有许多形式（如字节、字、双字、字符串），VxD 不仅要决定设备的拥有权，而且还要决定对申请的操作作出适当的响应。为了支持这些需要，VMM 将描述申请操作的信息作为回调协议的一部分进行传递。

图 7-2 说明了当 VMM 在捕获阶段调用争用处理器时如何设置寄存器数据。Port_Trap_Handler 例程可以决定哪个端口被访问，什么样的访问数据类型被申请，哪个 VM 提出申请，以及在必要时修改 VMM 用来恢复控制的处理器状态。在 ECX 中的值让处理器决定访问的确切类型。表 7-1 中给出了在 ECX 中可能的值。

```
回调输入：
EAX=输出命令的数据
EBX=当前 VM 处理
ECX=IO 类型
EDX=端口号
EBP=客户寄存器的结构
回调输出：
EAX=IN 命令返回的数据
```

图 7-2　VMM 建立寄存器数据

表 7-1　ECX 中可能的值

Byte_Input	000H	Dword_Output	014H
Byte_Output	004H	String_IO	020H
Word_Input	008H	Rep_IO	040H
Word_Output	00cH	Addr_32_IO	080H
Dword_Input	010H	Reverse_IO	100H

7.2.2 仿真多字节 I/O

对每个不同类型的 I/O 访问要调用 Port_Trap_Handler，它们有输入、输出、字节、字和双字。一个端口映射的 VxD 必须正确响应所有类型的端口访问。但是它们大多数是由设备字节导向的，所以通过仿真支持非字节的访问有意义。VMM 提供支持这种仿真的服务。实际上，如果一个设备支持字或双字的 I/O，它的处理器就应该直接支持这种模式，而不是通过仿真。

为了在字节导向的处理器中处理多字节访问，只要处理器遇到字或双字的访问，它就要调用 VMM 的服务 Simulate_IO（）。Simulate_IO（）将一个字分为两个字节，使处理器调用字节访问；而将双字分为 4 个字节，使处理器调用字节访问。Simulate_IO（）还能分解字符串 I/O，通常结合 REP 前缀的命令 INSB/OUTSB 来实现。

VMM.INC 提供了一个宏 Emulate_Non_Byte_IO，使得处理非字节访问更容易。这个宏执行为字节访问测试 ECX 的程序，如果 ECX 指示是非字节访问就跳到 Simulate_IO（），如果 ECX 指示是字节访问就不管。这个宏必须是处理器的第一个命令，如下所示。

```
Port_Trap_Handler:
Emulate_Non_Byte_IO     ; 跳到Simulate_IO或不管处理字节访问
cmp[VM_Owner],0.
```

PORTTRAP 使用这个宏处理除了字节访问之外的所有事情。

一旦处理器将访问简化成字节导向的访问，并解决了所有权的问题时，它必须完成 I/O 操作。如果申请了一个输入操作，那么结果字节应该在 EAX 中返回。VMM 将把 EAX 中的内容移到 CRS。如果申请了一个输出操作，EAX 中的值应该送到相应的目的地。例子 PORTTRAP 假定对选择的端口进行简单地读写访问，在 Port_trap_Handler 的最后几行实现，如下所示。

```
do_io:
      jecxz  do_read
      out    dx,al
      ret
do_read:
      in     al,dx
      ret
```

1．检测检验所有权

Port_Trap_Handler 在使用宏 Emulate_Non_Byte_IO 处理非字节访问之后，就执行它的判优程序。程序首先检测参数 VM_Owner 以决定 I/O 设备是否有占用者。如果设备有占用者，而且当前的 VM 就是占用者，Port_Trap_Handler 就像前面讲的那样执行 I/O 命令。如果没有占用者，Port_Trap_Handler 就调用一个内部的命令 Set_Owner 在 EBX 中传送占有者的 VM 控制。Set_Owner 在 EBX 中设置 VM 控制的参数 VM_Owner，并且禁止新的

占有者 VM 的“本地”获取。在禁止本地获取后，占有者 VM 可以访问设备而不会导致错误发生，而且这样不会从 PORTTRAP 产生冲突。其他 VM 的访问会产生一个错误以及对 Port_Trap_Handler 的调用。在调用 Set_Owner 后，Port_Trap_Handler 执行一个新占有者的 I/O。

Set_Owner 使用 VMM 的服务 Disable_Local_Trapping 来禁止新占有者 VM。这个服务从 EBX 中取得 VM 控制的参数，从 EDX 中取得端口号。Set_Owner 在循环中调用这个服务，通过每次改变端口号来禁止获取每个设备的端口。

如果另一个 VM 已经占有了设备，Port_Trap_Handler 就调用服务 SHELL_Resolve_Contention（）来询问用户哪个 VM 应该被指定为占有者。不过，PORTTRAP 并不是每一次而只在第一次，因为非占有者 VM 的访问打搅用户。

这是在每个 VM 的控制块中通过使用 CB_Flags 字段完成的。在调用 SHELL_Resolve_Contention（）之前，Port_Trap_Handler 首先检测 PORTTRAP 的每个 VM 数据中的一位（Contented_Bit）。如果这位被置，说明 POTYTRAP 已经警告过用户这个 VM 正在访问端口，并且要求指定一个占有者。这时 Port_Trap_Handler 不指定占有者，而且不执行 I/O 访问。相反，处理器在返回时 AL＝0xFF，它将显示在当前的 VM I/O 地址上没有设备。

如果 Contented_Bit 没有被置，Port_Trap_Handler 就设置它并使用 SHELL_Resolve_Contention（）询问用户哪个 VM 被指定为占有者。

SHELL_Resolve_Contention（）服务任务当前占有者的 VM 控制在 EAX 中，当前的 VM 在 EBX 中，以及 8 字节的地址、设备名在 ESI 中。SHELL VxD 显示一个列有每个 VM 名称和设备名称的对话框，用户从中选择占有设备的 VM。

SHELL 服务在 EBX 中返回用户的选择，Port_Trap_Handler 将 VM 控制传送给 Set_Owner。Set_Owner 更新 VM_Owner，并且对占有者 VM 禁止捕获。

2．VM_Terminate 处理器

一旦 VM 获得设备的所有权，它就会一直占有到 VM 结束或者用户通过 SHELL_Resolve_Contention（）服务选择另一个占有者。为了检测第一种情况，PORTTRAP 处理 VM_Terminate 消息。VM_Terminate 程序检测销毁的 VM 是不是设备的占有者，如果是，就将 VM_Owner 设置为 0 来标注设备没有被占有。VM_Terminate 不需要再允许对 VM 的端口捕获，因为 VM 已经被销毁。

7.2.3　安装和使用 PORTTRAP

在 Windows X 下，VxD 在 Windows 开始时通过 SYSTEM.INI 中的“device=”加载。Windows 不通过动态加载或卸载 VxD 的机制。要加载 PORTTRAP.386，在 SYSTEM.INI 的[386Enh]中加入“device=porttrap.386”。

测试 PORTTRAP 的简单方法是打开几个 DOS 框，并且使用 DEBUG 访问端口 400H～403H。当你第一次试图在不同的 DOS 框中访问端口，访问端口后将会在一个 DOS 框中得到 Device Contention 的对话框。如果你指定最初的 DOX 框为占有者，后面第二个 DOS

框的的访问就不导致对话框。但是如果打开第三个 DOS 框，并从这里访问端口，就会再次看到 Device Contention 的对话框。

7.3 例子 PAGETRAP

内存映射的设备也需要 VxD 执行设备判优。因为标准的 PC 内存映射的设备是图像适配器，并且图像设备驱动程序（VDD）已经虚拟了它，所以现在设计的 PAGETRAP 是虚拟单色图像适配器。单色适配器是非标准的，不过许多开发人员都需要有一个对于在调试中能够使用的程序。

7.3.1 初始化例程

为截取对内存设备的访问，PAGETRAP 在 Device_Init 消息处理器中调用以下的 VMM 服务：

1．Assign_Device_V86_Pages（）告诉 VMM，设计的 VxD 要使用 V86 线性地址空间中特定范围的页。

2．ModifyPageBits（）将页标注为不存在的，所以在 VM 访问页时就会产生一个页错误。

3．Hook_V86_Page（）为这些页注册一个页错误处理器。

注意 PAGETRAP 不分配物理内存的页，因为内存已经由设备提供。

VMM 内存服务与其他 VMM 服务不相同，因为它们通过栈而不是寄存器传递参数。PAGETRAP 通过宏 VMMcall 来调用这些服务，VMM 调用宏时自动将参数压入栈中。

VMM 服务 Assign_Device_V86_Pages（）的参数在图 7-3 中列出。VMHandle 是“0”，说明分配是局部的（只对某个 VM 而言），这个约定允许两个 VxD 在都使用一个局部分配时可以在两个不同的 VM 中声明相同的页；如果两个 VxD 试图全局声明相同的页，VMM 就会返回一个错误。PAGETRAP 使用单色图像适配器的物理地址和地址范围来为参数 VMLinrPage 和 nPages 派生相应的值。由于单色适配器映射到物理地址 B0000H，并且使用的大于单个 4KB 页，所以 VMLinrPage 参数是 B000H/4096（B0H），而 nPages 的值是“1”。这个服务的 flags 参数一直是“0”。

```
Assign_Device_V86_Pages (VMLinrPage, nPages, VMHandle, flags);
VMLinrPage          : 页号（在 0 到 0x10F 之间）;
nPages              : 页（4K）的数量;
VMHandle            : VM 处理, 如果分配是局部的, 就为 0;
Flags               : 必须为 0。
```

图 7-3 Assign_Device_V86_Pages（）的参数

VMM 的服务 ModifyPageBits（）的参数在图 7-4 中列出，它被 Device_Init 服务调用

来标注单色适配器的页在 System VM 中不存在。PAGETRAP 还要在它的 Create-VM 消息处理器中调用这个服务，以便在页表中为所有新的 VM 将单色适配器的页标注为不存在。

```
    ModifyPageBits
(VMHandle, VMLinPgNum, nPage, bitAND, bitOR, pTYPE, flags);
    VMHandle            ; 页位被调整的 VM 的句柄;
    VMLinrPage          ; 页号（在 0 到 0xFF 之间）;
    nPages              ; 页（4K）的数量;
    bitAND              ; 当前的页为与这一位的值相与;
    bitOR               ; 当前的页为与这一位的值相或;
    pType               ; 为 PG_HOOKED 或者 PG_IGNORE;
```

图 7-4　ModifyPageBits（）的参数

PAGETRAP 给 VMHandle 的值（VMLinPgNumhe nPages）是自解释的，参数 BitAND 和 bitOR 的值的每一位与机器页表入口的各位完全匹配。为了强制一个页错误，PAGETRAP 将清除 P_PRES，P_WRITE 和 P_USER 位。这个位的特殊组合是公用的，而且 VMM.INC 为组合定义了 P_AVAIL 等式。所以 PAGETRAP 的 Device_Init 程序中参数 bitAnd 的值是 NOT P_AVAIL，参数 bitOR 的值是“0”。参数 pType 的值是 PG_HOOKED，因为 DDK 文件上如果 P_PRES 被清除，那么 PG_HOOKED 就必须被使用。

7.3.2　错误处理器例程

PAGETRAP 使用和 PORTTRAP 一样的设备判优策略（即接触它，就占有它），而且判优也与 PORTTRAP 一样，在错误处理器中进行。PAGRTRAP 更简单，它不要求用户解决争用问题。这说明 PAGETRAP 不使用每一个 VM 数据，也不需要分配在 CB 中的空间。在占有者 VM 被指派后，PAGETRAP 导致所有的占有者 VM 访问内存映射的设备来更新设备，所有的非占有者访问不是被忽略（写）就是返回 FFH（读）。

要得到这个设想的行为，PAGRTRAP 使用 PhysIntoV86（）服务更新页表，使一定范围内的 V86 线性地址空间映射到一定范围内的物理内存。图 7-5 给出了 PhysIntoV86（）参数。

```
PhysIntoV86 (PhysPage, VMHandle, VMLinPgNum, nPages, flags);
PhysPage            ; 物理地址空间的页号（0—FFh）;
VMHandle            ; 发生映射的 VM 句柄;
VMLinPgNum          ; 线性地址空间页号（0—FFh）;
nPages              ; 页（4K）的数量;
```

图 7-5　PhysIntoV86（）的参数

Page_Fault_Hanbler( )使用 EBX 中的错误 VM 的句柄作为 VMHandle 的参数。PhysPage 和 VMLinPgNum 都等于“0”，nPages 等于“1”。在调用 PhysIntoV86（）后，单色适配器页的线性地址等于物理地址，VM 对单色适配器的线性地址的读和写直接到设备，而

不会导致页错误。如果设备没有占有者，这个动作就由 Page_Fault_Handler 执行。另一方面，如果设备硬件有了占有者（一个 VM 已经访问了页并且被指定了所有权），PAGETRAP 就可以执行其他动作。PAGETRAP 希望通过对 VM 地址的读取返回 0xFF，使在那个地址的设备看起来好像不存在。但和端口捕获处理器不一样，VxD 的页错误处理器没返回值，它可以用来对某个特定的内存读访问返回 0xFF。

PAGEPORT 有两个选择，VM 看不到物理地址 B000H 处的设备。一个是一段物理地址 PageAllocate（），将它填充为 FFH，当页错误发生时就将 VM 的页映射到被分配的填充 0xFF 的页。虽然可以写内存，但是 VM 看到的是读 FFH 的那段内存。如果设备在这页实际存在，那么读的就不是 FFH，并且 VM 决定设备是不存在的，不允许后面的访问。另一种选择是减少对 VxD 的位操作。我们可以将 VM 的页映射到特殊的已经被 VMM 分配过的被称为 nul page 的页，nul page 在不同时间被映射到不同位置，所以它的内容是随机的。这个行为也需要 VM 来决定设备是不存在的，并不允许后面的访问。以经验之谈，nul page 经常映射到不存在的 RAM，所以读回的结果是 FFH。

PAGETRAP 使用空页的方法。如果设备已经被占有，Page_Fault_Handler 首先调用 GetNulPageHandle（）来返回 VMM 对空页的控制。接着 PageFault 调用 VMM 的服务 MapIntoV86（），将 VM 的线性地址空间映射到物理内存的 nul page。图 7-6 给出了 MapIntoV86（）的参数。

```
    MapIntoV86
(hMem, PhysPage, VMHandle, VMLinPgNum, nPages, PageOff, flags);
    HMem            : 由 PageAllocate 或 GetNulPageHandle 返回的控制;
    PhysPage        : 物理地址空间的页号 (0—FFh);
    VMHandle        : 发生映射的 VM 句柄;
    VMLinPgNum      : 线性地址空间页号 (0—FFh);
    nPages          : 页 (4K) 的数量;
    PageOff         : 在 hMem 块中被映射在 VMLinPgNum 的页;
```

图 7-6　PhysIntoV86（）的参数

PAGETRAP 使用从 GetNulPageHandle（）返回的结果作为 hMem 的值，使用在 EBX 中传递的错误 VM 的句柄作为 VMHandle 的值。PhysPage 和 VMLinPgNum 都等于 B0H，使得线性地址直接映射到物理地址。NPages 是"1"，PageOff 是"0"，所以 NUL 页的第一块映射到 B0H。在调用 MapIntoV86（）后，VM 对单色适配器线性地址的读和写直接去 nul page，而不会产生页错误。

7.3.3 结束处理器

PAGEPORT 还要处理消息 VM_Terminate。VM_Terminate 处理器检测销毁的 VM 是否是设备的占有者。如果是，就将 VM_Owner 参数设为"0"来将设备标注为未占用。因为 VM 已经被销毁，对 VM 的页表做的任何事都不必要。

7.3.4 测试 PAGETRAP

对 PAGETRAP 进行测试，可以通过打开逆光 DOS 命令窗口使用 DEBUG 来对地址 B000：0000H 进行读或写操作，用单色适配器来测试 PAGETRAP。如果你已经安装了适配器，就能够在打开的第一个 DOS 窗口中用 DEBUG 对地址 B000：0000H 读或写，但是不是第一窗口。在其他的 DOS 窗口读或写时，窗口中看到的是随机数。如果根本没有适配器，那么从第一个 DOS 窗口读到的只是 FFH，而从其他 DOS 窗口读到的是随机数。

7.4 总 结

用 VxD 的组件和 VMM.INC 宏所给的样板对建立捕获端口或内存访问的 VxD 来说几乎是直接的。不管它们是否简单，这样的捕获 VxD 不仅对设备判优，而且对于测试和调试常规的设备驱动程序，Windows 应用程序和可以运行在 DOS 框中的 DOS 应用程序都非常有用。

这些判优 VxD 很简单，因为它们处理 VMM 消息的最小化，管理一般的设备，不处理中断，而且不为应用程序和其他 VxD 提供特殊的服务。第 8 章的 VxD 通过加中断服务和应用程序接口来扩展这个基本结构。

列表 7.1 porttrap.asm

```
; ===========================================================
        .386p
; ===========================================================
; *****************************************************
;        INCLUDES
; *****************************************************
.XLIST
        include VMM.inc
        include VPICD.INC
        include Debug.INC
        include SHELL.INC
.LIST
; *****************************************************
;    VIRTUAL  DEVICE  DECLARATION
; *****************************************************
Declare_Virtual_Device   PORTTRAP ,1 ,0 ,PORTTRAP_Control,
Undefined_Device_ID ,Undifined_Init_Order
; *****************************************************
; Initialization  data
```

```
VxD_IDATA_SEG

VxD_IDATA_ENDS
; ******************************************************
; Global data
VxD_DATA_SEG

Num_Io_Ports     dw       3
IO_Base         dw       0400h
VM_Owner        dd       0
CB_Offset       dd       0
Device_Name      db       'P', 'O', ' R', 'T', 'T', 'R', 'A', 'P'

ControlBlock  STRUC
CB_Flags        dw     0
ControlBlock  ENDS

Contended_Bit    EQU     0

VxD_DATA_ENDS

VxD_LOCKED_DATA_SEG

VxD_LOCKED_DATA_ENDS
; ******************************************************
; Initialization  code
VxD_ICODE_SEG

;   PORTTRAP_Device_Init
;   DESCRIPTION:
;   ENTRY:        EBX  is  SYS  VM  handle
;   EXIT:
;   USES:
BeginProc  PORTTRAP_Device_Init

        MovZx      ecx,Num_Io_Ports
        MovZx      edx,Io_Base
        Mov        esi,OFFSET32  Port_Trap_Handler
Hook_port:
        VMMCall    Install_IO_Handler
        Jnc        next_port
        Debug_out  'Failed  to  hook  port  #dx'
```

```
        Stc                             ; fail  VxD  load
        Ret
Next_Port:
        Inc         edx
        Loop        hookport
        Mov         exa,SIZE  ControlBlock
        VMMcall      Allocate_Device_CB_Area,<eax,0>
        Test        eax,eax
        Jnz         ok_cb
        Debug_Out  'Failed  to  alloc  CB'
        Stc                             ; fail  VxD  load
        Ret
Ok_cb:
        Mov         [CM_Offset],eax
        Clc                             : sucessfull  VxD  load
        Ret

EndProc  PORTTRAP_Device_Init

VxD_ICODE)ENDS
; ****************************************************
VxD_CODE_SEG

;   PORTTRAP_Control
;   DESCRIPTION:  dispatch control messages to correct handlers
;   ENTRY:
;   EXIT:   Carry  clear  if  no  error
:   USES:
BeginProc  PORTTRAP_Control

     Control_Dispatch   Device_Init,  PORT_Device_Init
     Control_Dispatch   VM_Terminate,  PORT_VM_Not_Executeable
     Clc
     ret

EndProc  PORTTRAP_Control

; ****************************************************
;   PORT_VM_Not_Executeable
:   DESCRIPTION:
:   ENTRY:        EBX  is  SYS  VM  handle
:   EXIT:
```

```
;  USES:
BeginProc  PORT_VM_Not_Executeable.

       Cmp     ebx, [VM_Owner]       ; is VM terminating the device owner?
       Jne     vm_not_owner          ; no,nothing to do
       Mov     [VM_Owner],0          ; yes,now device has no owner
Vm_not_owner:
       Clc
       Ret

EndProc  PORT_VM_Not_Executeable

; ***************************************************
;   Port_Trap_Handler
;   DESCRIPTION:
;   ENTRY:    EAX=data for OUT instraction
;             EBX=current VM handle
;             ECX=IO  type
;             EBX=port  number
;             EBP->Client  Register  Structure
;   EXIT:     EAX=data for IN instuction
;   USES:
BeginProc  Port_Trap_Handler

       Emulate_Non_Byte_IO

       Cmp      [VM_Owner],0              ; does device hav an owner?
       Je       do_io                     ; no,then assign new one

       Cmp      [VM_Owner],ebx
       Je       do_io

       Mov          eax,[CB_Offset]
       Add          eax,ebx
       Bts          [eax,CB_Flags],Contented_Bit
       Jnc          raise_contention
       Mov          al,0FFh
       Ret
raise_contention:
       mov          eax,[VM_Owner]      ; EAX=VM owner
       lea          esi,[Device_Name]   ; esi->device name
; EBX=VM  contender
```

```
        VxDcall     SHELL_Resolve_Contention
        Xor         ebx, ebx
        Call        Set_Owner          ; set to no owner
        Mov         ebx, eax           ; EAX=new VM owner
New_owner:
        Call        Set_Owner
Do_io:
        Jecxz       do_read
        Out         dx, al
        Ret
Do_read:
        In          al, dx
        Ret

EndProc  Port_Trap_Handler
; ****************************************************
Beginproc  Set_Owner

        Movzx       edx, Io_Base
        Movzx       ecx, Num_Io_Ports

        Cmp         ebx, 0
        Je          enable_trap
Disable_trap:
        VMMcal      Disable_Local_Trapping
        Inc         edx

        Loop        disable_trap
        Jmp         SHORT  save_owner
Enable_trap:
        Mov         eax, ebx
        Mov         ebx, [VM_Owner]
        VMMcal      Enable_Local Trapping
        Inc         edx

        Loop        enable_trap
        Mov         ebx, eax
Save_owner:
        Mov         [VM_Owner], ebx
        ret

Endproc  Set_Owner
```

```
VxD_CODE_ENDS
END
; End of file
```

列表 7.2　PORTTRAP.DEF

```
LIBRARY       PORTTRAP
DESCRIPTION    'Win386 PORTTRAP Device(Version1.0) '
EXETYPE       DEV386
SEGMENTS
             _LTEXT  PRELOAD  NONDISCARDABLE
             _LDATA  PRELOAD  NONDISCARDABLE
             _ITEXT  CLASS  'ICODE'  DISCARDABLE
             _IDATA  CLASS  'ICODE'  DISCARDABLE
             _TEXT   CLASS  'PCODE'  NONDISCARDABLE
             _DATA   CLASS  'PCODE'  NONDISCARDABLE
EXPORTS
             PORTTRAP_DDB   @1
```

列表 7.3　PAGETRAP.ASM

```
; ==========================================================
       .386p
; ==========================================================
; ****************************************************
;        INCLUDES
; ****************************************************
.XLIST
       include VMM.inc
       include VPICD.INC
       include Debug.INC
       include SHELL.INC
.LIST
; ****************************************************
;     VIRTUAL  DEVICE  DECLARATION
; ****************************************************
Declare_Virtual_Device   PAGETRAP ,1 ,0 ,PAGETRAP_Control,
Undefined_Device_ID ,Undifined_Init_Order
; ****************************************************
; Initialization  data
VxD_IDATA_SEG

VxD_IDATA_ENDS
```

```
; *****************************************************
; Global data
VxD_DATA_SEG

PageNumber        EQU   0B0000h/4096
VM_Owner          dd    0

VxD_DATA_ENDS

VxD_LOCKED_DATA_SEG

VxD_LOCKED_DATA_ENDS
; *****************************************************
; Initialization  code
VxD_ICODE_SEG

;   PAGETRAP_Device_Init
;   DESCRIPTION:
;   ENTRY:        EBX  is  SYS  VM  handle
;   EXIT:
;   USES:
BeginProc  PAGETRAP_Device_Init
       VmmCALL    _Assign_Device_V86_Pages,<PageNumber,1,ebx,0>
       Jnz        assign_ok
       Debug_Out  'Assgn_Device_V86_Pages  failed'
       Jmp        SHORT  init_err
Assign_ok:
       Mov        eax,PageNumber
       Mov        esi,OFFSET32  Page_Fault Handler
       VMMcall     Hook_V86_Page
       Jnz        hook_ok
      Debug_Out  'Hook_V86_Page  failed'
      Jmp         SHORT  init_err
Hook_ok:
       VMMcall  _ModifyPagebits,<ebx,PageNumber,1,<not
P_AVAIL>,0,PA_HOOKED,0>
      Jnz     Modify_ok
      Debig_Out  'Modify_Page_Bits  failed'
      Jmp        SHORT  init_err
Modify_ok:
      Clc            ; cucessfull  VxD  load
      Ret
```

```
Init_err:
      Clc
      Ret

EndProc  PAGETRAP_Device_Init

VxD_ICODE_ENDS
; ****************************************************

VxD_CODE_SEG

; ****************************************************
;   PAGETRAP_Control
;   DESCRIPTION:  dispatch control messages to correct handlers
;   ENTRY:
;   EXIT:   Carry  clear  if  no  error
;   USES:
BeginProc  PAGETRAP_Control

     Control_Dispatch   Device_Init,        PORT_Device_Init
     Control_Dispatch   VM_Not_Executeable, PORT_VM_Not_Executeable
     Clc
     ret

EndProc  PAGETRAP_Control

; ****************************************************
;   PAGE_VM_Not_Executeable
;   DESCRIPTION:
;   ENTRY:        EBX  is  SYS  VM  handle
;   EXIT:
;   USES:
BeginProc  PAGE_VM_Not_Executeable

       Cmp     ebx,[VM_Owner]    ; is VM terminating the device owner?
       Jne     vm_not_owner      ; no,nothing to do
       Mov     [VM_Owner],0      ; yes,now device has no owner
Vm_not_owner:
       Clc
       Ret

EndProc  PAGE_VM_Not_Executeable
```

```
; ****************************************************
;   PAGE_Fault_Handler
;   DESCRIPTION:
;   ENTRY:        EAX=faulting  page  number
;                 EBX=current  VM  handle
;   EXIT:
;   USES:
BeginProc  PAGE_Fault_Handler

        Cmp      [VM_Owner],0
        Je       new_owner

        VMMcall  _GetNulPageHandle
        VMMCall  _MapIntoV86,<eax,pageNumber,ebx,PageNumber,1,0,0>
        Jnz      map_ok
        Debug_Out  'MapIntoV86  failed'
Map_ok:
       Ret

New_owner:  .
        Mov      [VM_Owner],ebx
        VMMCall  _PhysIntoV86,<pageNumber,ebx,PageNumber,1,0>
        Jnz      phys_ok
        Debug_Out  'PhysIntoV86  failed'
Map_ok:
       Ret

EndProc  PAGE_Fault_Handler

VxD_CODE_ENDS

END
; End of File
```

列表 7.4　PAGETRAP.DEF

```
LIBRARY        PORTTRAP
DESCRIPTION    'Win386  PAGETRAP  Device(Version1.0) '
EXETYPE        DEV386
SEGMENTS
               _LTEXT  PRELOAD  NONDISCARDABLE
               _LDATA  PRELOAD  NONDISCARDABLE
```

```
            _ITEXT  CLASS  'ICODE'  DISCARDABLE
            _IDATA  CLASS  'ICODE'  DISCARDABLE
            _TEXT   CLASS  'PCODE'  NONDISCARDABLE
            _DATA   CLASS  'PCODE'  NONDISCARDABLE
EXPORTS
            PAGETRAP_DDB   @1
```

第 8 章　服务硬件中断的 VxD

Windows 将硬件中断转换成可以触发各种虚拟机中的 ISR 执行事件的特性，是虚拟环境中最容易造成混乱的原因之一。Windows 必须明确不仅要相关的虚拟设备知道中断，而且还要保证相关的虚拟机也知道中断。这个过程不仅复杂，而且涉及到大量上层的东西，Windows 的 ISR 的时间可以是 DOS 的 ISR 的 20 倍。

实际上，中断可以触发级联 4 个级别代码的活动，分别如下：

- VMM 中到一个程序的处理器向量
- VMM 在一个或多个 VxD 中调用注册过的处理器
- VMM 为保护模式处理器仿真中断
- VMM 为 V86 模式处理器仿真中断

除了第一级别外，程序员可以在其他级别安装中断处理器（实际上由于 VxD 在 0 层运行，所以它也可以在实际的 IDT 中进行第一级安装，但是 Microsoft 反对这样做）。第 4 章中的 ISR 作为一个在 VM 内部运行的的处理器来安装，由 DOS 应用程序安装的处理器被限定为 V86 模式处理器。本章将解释中断机制，怎样编写一个作为 VxD 的一部分进行安装的处理器，以及怎样编写在虚拟机中安装处理器的 VxD。

8.1　中断和 VMM

在扩展模式中，处理器运行在 3 个不同的状态。当 DOS 应用程序执行时，处理器运行在 V86 模式中（在 V86 VM 中）；当 System VM 执行或者 DOS VM 已经切换到保护模式时，处理器运行在第 3 层保护模式中；当 VMM 或者 VxD 执行时，处理器运行在第 1 层保护模式中。

如果不考虑处理器的当前状态，当一个硬件中断发生时，处理器切换到第 0 层的保护模式。接着处理器在 IDT 中找到中断处理器的地址，并开始运行它。这不是 Windows 的规定，而只是 80X86 机器的工作方法。

然而，如图 8-1 所示，Windows 不使用 IDT 查找中断处理器。相反，Windows 将所有的 IDT 入口指向 VMM 中的一个例程。由 VMM 例程来确定它是作为异常事件的结果，还是作为中断的结果而被调用。VMM 自己管理异常事件，而将所有的硬件中断交给被称为 VPICD(Virtual Programmable Interrupt Controller Device)的 VxD 处理。如果一个 VxD 为中断进行了注册，VPICD 就将这个中断交给它处理，否则 VPICD 将中断交给 VMS 中的 reflection 处理。

VxD 通过调用 VPICD 的服务 VPICD_Virtualize_IRQ（）来为硬件中断注册，并将回调例程的地址交给 VPICD。一旦 VxD 进行了注册，它就可以作为一个实在的中断处理器

自己处理硬件中断，或者通过调用 VPICD 服务 VPICD_Set_Int_Request（），将中断映射到一个 VM。也就是说，VxD 让 VM 处理器代替它进行处理。

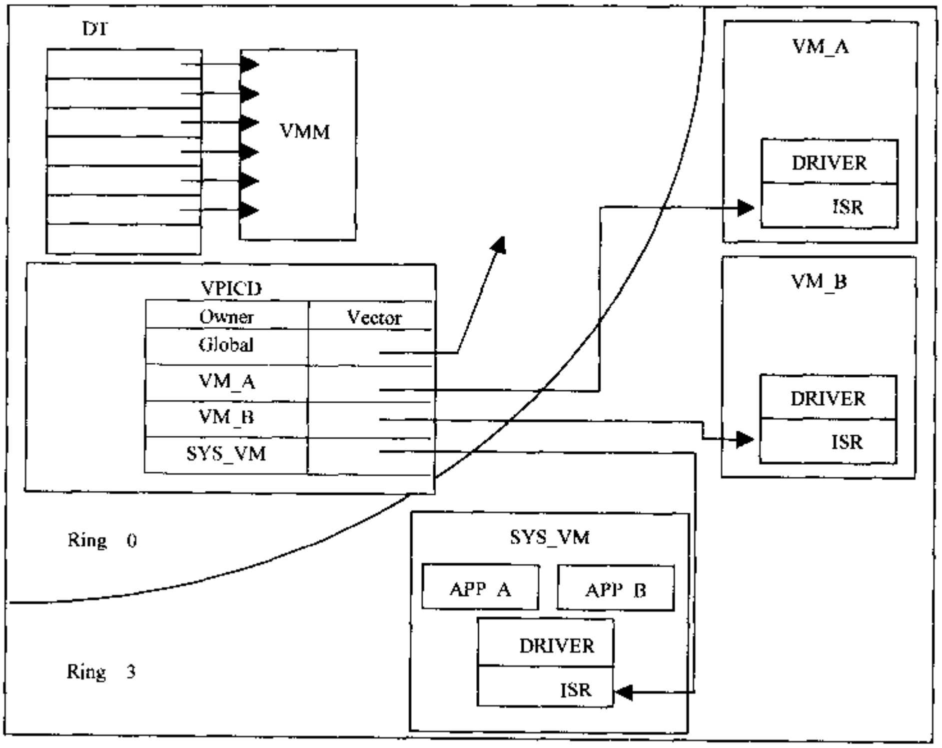

图 8-1　VPICD 和中断关系图

如果没有 VxD 为中断注册，VPICD 就不服务于中断，因为它没有关于设备的有关知识。反之，VPICD 就一直映射中断到 VM。这样做有两个原因：注册的 VxD 决定影射它，或者没有 VxD 为中断注册。

8.1.1　中断反射的过程

由于硬件中断随时都可能发生，所以 VM 可以在 VPICD 要映射一个中断到 VM 的时候执行。作为映射的第一步，VPICD 必须迫使想得到的 VM 被调度。VPICD 通过在最高优先级 Time_Critical_Boost 调用 VMM 服务 Call_Priorty_VM_Event（）来迫使调度改变。

VPICD 通过这个服务提供回调，所以 VMM 可以在目标 VM 被调度时通知 VPICD。VPICD 使用另一个 VMM 服务 Simulate_Int( )响应回调来修改 VM 的执行环境。Simulate_Int（）改变 VM 的状态信息，以便它在刚结束一个 INT 命令的执行后出现：VM 的 CS，IP 和标志寄存器被压入 VM 的栈；VM 新的 CS 和 IP 值从 VM 的 IVT 中取得（在 VM 地址空间的位置是 0000:0000）。当 VPICD 从这个回调中返回并且 VMM 切换回 V86 模式时，VM 为硬件中断执行中断处理器。

8.1.2 哪个 VM 获得中断

如果 VxD 反射中断是因为没有 VxD 为它注册，VPICD 必须决定哪个 VM 获得中断。为做出这个决定，VPICD 对局部中断和全局中断有所区别。局部中断在 Windows 启动阶段是禁止的（在物理的 PIC 中），而全局中断在 Windows 启动阶段被允许。注意：由于全局中断是允许的，所以在 Windows 开始时全局中断必须已经安装了中断处理器——在 BIOS 中，一个 DOS 驱动程序或者一个 TSR。下面简单研究一下这个论述的重要性。

在 Windows 初始化后，一个 VM 可能安装一个中断处理器并且在 PIC 中允许它。由定义可知，它是一个局部中断。现在 VPICD 将允许中断的 VM 作为它的占有者，从此 VPICD 一直将这个中断反射到占有者 VM。这是有道理的，因为 VM 中断处理器只存在于这个 VM 中，如果反射它到其他的 VM 将会导致调用假的地址。

另一方面，全局中断没有占有者，但是在中断发生时，哪个 VM 产生了执行，它就被反射到哪里。它能工作是因为 Windows 开始时全局中断就被允许，因此当 Windows 开始时就有处理器被安装。也就是说，这个处理器存在于所有的 VM 中。这样，它几乎不必关心 VPICD 将中断反射到哪个 VM，每个 VM 都有一个 IVT 指向同一个处理器。

一旦 VPICD 选择了反射的 VM，它就一直选择调用保护模式处理器。只有在没有安装保护模式处理器时 VPICD 才选择 V86 模式处理器。VPICD 保留一个假的 IDT，它在一个保护模式应用程序通过 Dos Set Vector 或者 DPMISet Protected Mode Vector 安装中断处理器时被设置，这个假的 IDT 用来获得保护模式处理器的地址。类似地，VPICD 保留一个假的 IDT，它在一个 DOS 应用程序安装中断处理器（或者 Windows 应用程序调用 DPMI Set Real Mode Vector）时被更新，这个假的 IDT 用来获得 V86 模式处理器的地址。

如果有一个 VxD 为中断注册，VxD 计划将它反射到一个 VM，那么选择适当的 VM 和把中断指示到正确的处理器是 VxD 的责任。VxD 跟踪设备的拥有权并将中断反射到占有者 VM。VxD 通过调用 VPICD_Set_Int_Request（）将占有者 VM 的处理交给 VPICD。

8.1.3 扩展模式中的中断执行时间

由于多级反射过程，在硬件中断发生的时刻和 VM 处理器执行的时刻之间有相当多的延时时间。如果你正在写 Windows 中断处理器程序，而这个延时将导致设备发生问题，中断处理器就应该被 VPICD 直接调用而不是经过反射过程，将处理器直接放入 VxD（在本章中介绍）或者第 0 层的特殊的 DLL 处理器中（将在第三部分介绍）。

不幸的是，没有一个 VxD 可以保证能够实时响应中断。有两个因素使得在 Windows 下不可能实时响应，一个是由于机器本身的体系结构，另一个是由于在硬件中断和 VxD 处理器之间的层次数。例如，如果 VPICD 要将一个局部中断反射到一个 VM，VM 并没有在接收到中断时运行，那么 VMM 就必须暂停当前的 VM，再加载目标 VM，然后切换到保护模式或者 V86 模式下以恢复目标 VM 的执行。这些操作有耗时存在。

如果编写 VxD 服务于硬件中断，那么反应时间要比在实模式 DOS 下所期望的时间长。

例如当中断发生时，在 IDT 中已安装的中断处理器就被调用。这里就会不可避免地产生一些由体系结构占用的延时——从 V86 模式或第 3 层保护模式切换到第 1 层的中断处理器所用的时间。这只包含一个命令，而这个命令大概需要 200 个时钟周期执行。

在层切换之后，IDT 中的 VMM 处理器决定它被中断调用和非异常事件时，就调用 VPICD。VPICD 决定哪个 VxD 为中断注册，接着调用你的 VxD。这个 VMM-VPICD-VxD 过程又增加大约 80 到 100 个时钟周期，处理器在设备产生中断大约 300 个时钟周期后才被调用。相对于一些设备而言，这个响应时间不够快。

一旦处理器开始执行，它必须调用 VPICD_Physical_EOI（）直接将 EOI 发送到控制器，这需要用更多的周期。处理器返回到 VPICD 后，做一些清除工作。VPICD 返回到 VMM，做更多的消除工作。最后 VMM 执行 IRET 命令。IRET 完成另一个层转换，需要 200 或更多的时钟周期。所以开始运行 VxD 处理器大概需要 300 个时钟，并且可能需要 300 个时钟周期来结束执行并离开。如果设备需要高频率的服务，那么在设备产生下一个中断前就只有一小部分时间用来处理中断。

8.2 VxD 通信

VxD 有能力修改 VM 环境的一些组件，包括被 VM 占用的内存地址和 VM 的寄存器。有时这些修改可以在不需要 VM 时进行，例如当 VxD 截取 VM 对 I/O 端口的访问时，VxD 就修改 VM 的寄存器，使得 VM 在执行 IN 命令后看到的是 FF。有时这是很明确地进行，就像 VM 通过外部 API 调用 VxD，VxD 通过在 VM 的寄存器中返回的信息或修改 VM 地址空间的缓冲区来响应一样。

第 7 章已经介绍了两个 VxD，它们不需要 VM 的知识而通过端口捕获和页错误处理器来修改 VM 的环境。本章中的一个 VxD 介绍关于透明地修改的例子——迫使 VM 的中断处理器来执行，另一个 VxD 研究明确的方法——API。

8.2.1 传递参数

从应用程序的角度看，通过 API 与 VxD 通信就像是使用软件中断。应用程序（V86 或 PM）通过寄存器输入参数到 VxD，同时也通过寄存器接收输出的参数。然而在 VxD 结尾，这些参数并不在寄存器中出现，而是在 VMM 调用 VxD 的 API 入口点时，传递给 VxD 一个被称为“客户寄存器结构”（CRS）的专用数据结构，用来从应用程序获得参数。

CRS 是一个关键监控的数据结构，VMM 使用 CRS 保存 VM 寄存器的内容。在切换到第 0 层后，VMM 立即获取实际处理器寄存器内容的瞬时状态，并将它存储到被中断的 VM 的 CRS 中。之后，在切换回第 3 层前，VMM 利用 VM 的 CRS 中的内容再次加载实际处理器的寄存器。

但 CRS 不仅仅是作为在监控程序执行时 VM 寄存器内容的暂时存储器。VxD 可以通过检测和修改 VM 的 CRS 来检测和修改 VM 的寄存器内容，前面介绍的 VM CB 包含有

一个指向 CRS 的指针。另外，当 VxD 的 PM 或 V86 的 API 入口点被调用时，EBP 直接指向 CRS。

CRS 是一个简单的结构，它为每个处理器的寄存器保留一个字段。同时还有一些备用的字段，它们很少被 VxD 使用。这些备用的字段在 VM 处于保护模式中时保存 V86 的值，并在 VM 处于 V86 模式中时保存 PM 的值。

8.2.2 从应用程序调用 VxD 的 API

应用程序通过寄存器将参数传递到 VxD 时，与使用软件中断类似，只是实际调用机制有点不同。应用程序通过地址调用 VxD，而不是软件中断。但是这个 VxD 的地址首先通过执行一个软件中断得到。

为得到 VxD 的 API 地址，应用程序必须知道 VxD 的设备 ID 号。应用程序将 ID 号放入 BX 中，函数代码 1684H（获取设备 ID 号）在 AX 中，并执行软件中断 INT 2FH。从软件中断返回时，在 ES:DI 中包含的是应用程序用来直接调用 VxD 的地址，或者是因为某些原因调用失败时返回的 0:0（例如无效的 ID）。在得到入口点地址后，应用程序通过这个远程指针调用 VxD 的 API。

VMM 截取 INT 2FH，检测 ID 号，并且在 DDB 中寻找匹配的 ID。如果在 DDB 中没有包含指向相应的 API 入口点的指针，VMM 就向调用者返回一个错误（ES:DI＝0:0）；如果 DDB 包含指向 API 的指针，VMM 不返回这个指针到应用程序，因为这个指针是第 0 层的一个 32 位指针，不能被调用者使用。VMM 必须将这个第 0 层的地址转换成第 3 层的调用者才可以使用地址。

机器体系结构有一个被称为 call gate 的特性，它使用某层中的程序调用另一层中的程序。VMM 不使用这个特性。相反，VMM 返回一个包含非法指令（ARPL）的地址。当应用程序调用这个地址，非法指令将导致一个处理器错误，并且 VMM 获得控制。VMM 错误处理器决定其错误的原因并最终结束对 VxD 入口点的相应调用。

8.2.3 硬件中断处理 VxD

在扩展模式中，最快中断响应时间由直接服务中断的 VxD 提供。本节将检查从串口 COM1 服务于中断的 VxD 的 SVCINT（见列表 8-1～8-3）。SVCINT 还为 Windows 和 DOS 应用程序提供了一个 API。通过这个 API，应用程序可以提供给 VxD 要传送的数据或者要求 VxD 接收数据。在中断时，VxD 要么从它的传送层缓冲区传送数据，要么将接收到的数据放入它的接收层缓冲区。

SVCINT 通过消息 SyS_Critical_Init 安装它的中断处理器。VxD 中断控制器可以在某个初始化消息过程或者在稍后响应 VM 一个动作时被装入（例如访问端口或调用 VxD 的 API）。所使用的服务 VPICD_Install_Handler（）由 VPICD 提供。EDI 寄存器指向一个包含调用 VPICD_IRQ_Descriptor 所需的所有参数的结构，这个结构在表 8.1 中给出。

在这个结构中有许多字段，但是服务中断的 VxD 实际只使用开始 3 个：VID_IRQ_

Number，VID_Options 和 VID_Hw_Int_Proc。其他的字段被反射中断到 VM 的 VxD 使用。VIDOptions 是一个位图字段。Can_Share 位被设置来指示一个可共享的 IRQ，被清除来指示一个不可共享的 IRQ。如果可以共享，其他 VxD 也被允许虚拟这个 IRQ。当中断发生时，VPICD 就调用链中的每个处理器，直到返回 Carry 清除，此时中断服务被执行了。SVCINT 不共享 IRQ，就像大多数 PC 的串口不支持中断共享一样。

表 8-1　VPICD_IRQ_Descriptor 结构的字段

字　段	说　明
VID_IRQ_Number	被虚拟的硬件 IRQ 号（1 到 15）
VID_Options	设置 VID_Opt_Can_Share 的值，如果其他的 VxD 被允许虚拟 IRQ
注意：接着的回调被输入为 EAX=IRQ 控制，EBX=当前 VM 控制	
VID_Hw_Int_Proc	在硬件中断发生时回调：禁止中断。如果可共享，VxD 控制中断就清 Carry，否则设置 Carry
VID_Virt_Int_Proc	当 VPICD 将中断反射到 VM 时回调，与 VID_Virt_IRET_Proc 结合使用，在仿真的硬件中断的临界段执行。VxD 在 VID_Virt_Int_Proc 中设置标志 VM_servicing_int，在 VID_Virt_IRET_Proc 中清除它
VID_Virt_EOI	当 VM 发送给中断控制器一个 EOI 时回调，如果 VID_Hw_Int_Proc 因为不服务中断而没有发送 VPICD_Phys_EOI，此时 VPICD_Phys_EOI 应该在 VID_Virt_EOI 中发送
VID_Mask_Change_Proc	当 VM 在中断控制器中屏蔽或不屏蔽中断级别时使用，如果不屏蔽，ECX =0，否则 ECX!=0。当 VM 不屏蔽设备中断时，VxD 执行判优通常指定拥有权
VID_Virt_IRET_Proc	当 VPICD 将中断反射到 VM 时回调，参见 VID_Virt_Int_Proc
VID_IRET_Time_Out	在中断超时前 VPICD 允许的最大时间（ms）

SVCINT 文件中的 Hw_Int_Proc 带有它的中断处理器 SVCINT_Int 的地址。VID_IRQ_Number 是硬件 IRQ 号，而不是向量号，所以有效的 IRQ 号是从 0 到 15。SVCINT_IRQ_Desc 中的字段在 SVCINT 的数据段的结构确立中被静态地初始化。

SVCINT_Int 是 VxD 的中断处理器。这个处理器与第 4 章中非 VxD 的中断处理器非常相似，只是它用的是汇编。首先处理器读串口的中断标识寄存器（IIR）来确定中断的原因。如果接收数据是可用的，处理器就从串口读字节，并将它存入接收层缓冲区；如果串口已经准备好传送字节，处理器就把传送层缓冲区中的下一个字节送给它；如果这时传送缓冲区为空，它就关闭传送中断。

在返回前，SVCINT_Int 向中断控制器发送一个 EOI（End of Interrupt）信号来允许后面这个 IRQ 级别上的中断，DOS 或 Windows 中断处理器通过向控制器的命令寄存器写 EOI 值（20H）来完成它。VxD 永远不能写 EOI，但是可以使用服务 VPICD_Rhys_EOI（）来代替完成相同的目标。这个服务只有一个参数，即 IRQ 控制而不是 IRQ 编号。这个控制可以在进入 VxD 处理器的 EAX 中找到。

VxD 不能直接向控制器写 EOI 的原因是 Windows 使用中断控制器与 DOS 不同。VPICD

是中断通知的第一个 VxD，而且 VPICD 立即发送一个特定的 EOI 到控制器，对中断设备的中断级别是特定的。接着 VPICD 屏蔽（禁止）控制器上的这部分中断级别。有两个活动允许其他中断级别被认可，包括那些优先权比中断级别更低的。为了响应 VPICD_Phys_EOI（），VPICD 不屏蔽相同级别的中断。

8.2.4 API 调用服务

SVCINT 的 API 入口点是 SVCINT_API，通过 DDB 导出。这个函数首先从应用程序得到一些参数，进行有效性检测，然后使用在 AX 中传递的函数代码来调用子函数。这里有 4 个子函数：SVCINT_Open，SVCINT_Close，SVCINT_Read 和 SVCINT_Write。

所有在 VxD 和应用程序之间的通信都是通过 VM 的 CRS 来完成的 。函数代码存在 CRS 中的字段 Client_AX 中，并且用表达式[ebp.Client_AX]来访问。当调用函数 SVCINT_Read 或者 SVCINT_Write 时，应用程序传入两个参数：在 ES:BX 中的缓冲区指针和在 CX 中的长度。SVCINT_API 使用 VMM 服务 MapFlat（）将 V86 或第 3 层的缓冲区指针转换为第 0 层的地址。这个地址被临时存入 EDI 中，在这里可以被 SVCINT_Read 或者 SVCINT_Write 得到。

接着，SVCINT 检查调用 VM 的 VM 控制。如果函数是 SVCINT_Read，SVCINT_Write 或 SVCINT_Close，调用的 VM 必须与占有者 VM 相同（由 SVCINT_Open 指定）。如果这个条件没有满足，SVCINTClient_AX 在寄存器中将错误代码 SVCINT_API_Err_NotOwner 返回到 VM；如果函数是 SVCINT_Open，占有者 VM 必须是 0，这意味着再没有其他的 VM 打开端口；如果这个条件没有满足，SVCINT 在寄存器 Client_AX 中将错误代码 SVCINT_API_Err_AlreadyOpen 返回到 VM。如果所有的有效性检测都通过了，SVCINT_API 使用函数代码以及函数指针的表来调用相应的子函数。

SVCINT_Open 和 SVCINT_Close 都非常简单。SVCINT_Open 通过 EBX 将当前 VM 控制存储到它的变量 VM_Owner 中，SVCINT_Close 仅仅将 VM_Owner 置成 0。通过这个机制，SVCINT_API 就可以在占有者已经打开设备时防止其他的 VM 使用设备。

SVCINT_Read 和 SVCINT_Write 也很简单，使用的算法和第 4 章中非 VxD 中断驱动程序的函数 DeviceRead 和 DeviceWrite 相同。SVCINT_Read 从接收层缓冲区拷贝所需的字节数目（在 ECX 中传递）到调用者的缓冲区。SVCINT_Read 首先将 ESI 指向接收层缓冲区，将 EDI 指向调用者缓冲区，接着使用指令 MOVSB 拷贝一个字节数据。MOVSB 是一个功能很强的指令，可以将 DS:[ESI]中的字节拷贝到 ES:[EDI]，并自动将 ESI 和 EDI 加 1。SVCINT 不必设置 DS 或 ES 段，因为这两个段寄存器在任何 VxD 函数的入口寻址所有的 4GB 线性内存。LOOP 指令用来实现多次移动。LOOP 指令将 ECX 减 1，如果 ECX 不等于 0，就跳到循环的头部并获得下一个字节；如果 ECX 等于 0，就退出循环。SVCINT 不使用指令 REP MOVSB，因为它在每个字节移动时必须检测层缓冲区。

SVCINT_Write 与 SVCINT_Read 相对应，从调用者缓冲区拷贝所需数目的字节到传输层缓冲区。

8.3 硬件中断反射 VxD

很多时候，VxD 自己并不服务中断，而是通过使用 VPICD 的服务将它“反射”到 VM。VxD 可以将中断反射到它们选择的 VM，但是大多数 VxD 通过端口映射或页映射来指定设备的 VM 拥有权，或者通过 API，然后将所有的中断反射到占有者 VM。

例子 REFLECT 说明了怎样把中断反射到占有者 VM。例子程序虚拟了 COM1 中断（IRQ4），不过可以很容易地改变成其他的 IRQ 号。REFLECT 见列表 8.4～8.6。

当应用程序调用 VPICD_Virtualize_IRQ（）时，它向 IRQ_Descriptor 传送一个指针。它包含 5 个回调函数的指针，VPICD 使用这些回调通知 VxD 虚拟 IRQ 状态的改变。VPICD_IRQ_Descriptor 中的字段在表 8.1 中简要地进行了描述。这些回调对于将 IRQ 反射到占有者 VM 是很关键的。

REFLECT 在 Device_Init 阶段调用 VPICD_Virtualize_IRQ（）来虚拟 IRQ4。唯一的参数是指向 VPICD_IRQ_Descriptor 的指针 EDI，这个结构在 REFKECT 数据段中结构的声明中被静态地初始化。注意前面讨论的 SVCINT 只使用 VID…Hw_Int_Proc 回调，而 REFLECT 包括所有的 5 个回调。不同点在于 SVCINT 实际上服务中断，而 REFLECT 只将中断反射到 VM。VPICD_Virtualize_IRQ（）在 EAX 中返回一个 IRQ 控制，由 REFLECT 存储。这个控制在以后调用其他 VPICD 服务时使用。

8.3.1 回调 Mast

REFLECT 对拥有权的规则是：如果在 PIC 中允许中断，就能具有拥有权。当 VM 不屏蔽在中断控制器中的 IRQ4，REFLECT_Mask 就被调用。如果 VM 没有屏蔽，则 ECX 就为 0；否则 ECX 为非 0。如果活动没有屏蔽，REFLECT_Mask 就要检测变量 VM_Owner，以确定是否另外有一个 VM 占有了 IRQ。如果是，REFLECT_Mask 就使用服务 SHELL_Resolve_Contention（）来询问用户哪个 VM 要占有设备；如果当前没有 VM 占有 IRQ4，REFLECT_Mask 就在 EBX 中将 VM_Owner 设置为当前的 VM 控制，用以指定当前 VM 为占有者。

在返回前，调用 VPICD_Physically_Unmask（）服务来释放在实际控制器中的中断，这一步非常重要。如果没有 VxD 虚拟 IRQ，VPICD 就将所有的 IN 和 OUT 指令转移到中断控制器，并且不屏蔽在 VM 上的中断。但是一旦有 VxD 虚拟了中断，VPICD 就让开，而且 VxD 必须不屏蔽在 VM 上的中断。这个服务需要 EAX 中的 IRQ 控制，所以 REFLECT_Mask 将变量 REFLECT_IRQ_Handle 调入 EAX 中。

如果 VM 屏蔽 IRQ，REFLECT 就将 VM_Owner 设为 0，通过调用服务 VPICD_Physically_Mask（）向 VPICD 传送屏蔽申请，并且退出。没必要为响应屏蔽而将占有者设为“无”，因为在屏蔽时中断更变不能到达处理器。不过 VxD 应该在响应消息 VM_Terminate 时将占有者设为“无”，这样在响应屏蔽时非常有用，因为在用户退出程

序前，许多应用程序在它们结束时就关闭（屏蔽）了设备的中断。这时通过“不指定”拥有权，VxD 可以让另一个 VM 使用设备。

8.3.2　回调 Hw_Int

实际的反射过程发生在 REFLECT_Hw_Int 中，当指定在 IRQ4 上发生 VPICD 调用时。REFLECT_Hw_Int 可以反射（或者仿真）中断到占有者 VM，不过必须在下面的条件下：

1. 当前的（被中断的）VM 必须是占有者。
2. 当前 VM 的处理器没有服务设备中断。

REFLECT 使用 VM_Servicing_Int 标志来防止中断在 VM 处理前一个中断时被仿真到 VM。如果 VM 被太多的仿真中断所覆没，中断就将后进先出，而且 VM 中断处理器的堆栈就会溢出。VM_Servicing_Int 标志用指令 REFLECT_Virt_Int 和 REFLECT_Iret 来设置和清除，下面将会马上讲到。

如果上面的两个条件得到满足，REFLECT 就通过调用服务 VPICD_Set_Int_Request（）来将中断反射到 VM，IRQ 控制在 EAX 中，VM_Owner 在 EBX 中。注意当这个服务返回到 REFLECT_Hw_Int 时，VM 中断控制器并没有被调用。VPICD 只安排了一个稍后采取活动的事件，REFLECT_Hw_Int 在调用这个服务后就返回。

如果 REFLECT_Hw_Int 不反射中断，它就调用 Eat_Int（）。Eat_Int（）清除设备中的中断并通知 VPICD 向控制器发送一个 EOI。清除设备中的中断是重要的一步。如果设备的中断没有被清除，那么 IRQ 就仍然保留声明，而 VPICD 不会再看到从设备来的其他中断，因为 IRQ 是 edge_triggered。Eat_Int（）对串口来说很特别，因为它通过读取几个串口寄存器（中断认证寄存器、接收数据寄存器、行状态寄存器和 Modem 状态寄存器）来清除所有未决的中断。由于中断被实际服务，所以如果只是为了丢弃，Eat_Int（）也可以调用 VPICD_Phys_EOI（）通知 VPICD 向控制器发送 EOI。最后，Eat_Int（）返回到它的调用者 REFLECT_Hw_Int。

REFLECT_Hw_Int 一直伴随清除 Carry 返回到 VPICD。它通知 VPICD，IRQ 由 VxD 的设备引起，VPICD 不能调用在公用链路中的下一个 VxD。在写时，REFLECT 不共享中断，因为大多数 PC 的串口不需要硬件支持共享。如果设备完全支持共享 IRQ，可以很容易升级 VxD。用 REFLECT_Hw_Int_Proc 首先查询设备是否有悬而未决的中断，如果没有，就设置 Carry，以使 VPICD 调用链路中的下一个 VxD 处理器。

8.3.3　回调 EOI

REFLECT_EOI 在 VM 中断处理器（作为 REFLECT 调用 VPICD_Set_Int（）的结构被最终执行）向中断控制器发送 EOI 时被调用，REFLECT_EOI 代表 VM 调用 VPICD_Phys_EOI（）。VPICD_Phys_EOI（）使用的唯一参数是 EAX 中的 IRQ 控制。最后 REFLECT_EOI 加载 EBX 中的 VM_Owner 控制，并且调用 VPICD_Clear_Int_Request（）。这个服务通过 REFLEXT_hW_Int 为拥有者 VM 复位 IRQ 请求设置。如果没有这一步，VPICD

会立即再次调用 REFLECT_Hw_Int。

8.3.4 回调 Virt_Int 和 IRET

在 VPICD 每次开始将中断仿真到 VM 时，REFLECT_Virt_Int（）都要被调用，它标记 VM 中断处理器执行的开始。REFLECT_Virt_Int 设置 VM_Servicing_Int 标志来防止 REFLECT_Hw_Int 反射其他的中断到 VM，直到 VM 处理器返回为止。REFLECT 知道处理器什么时候返回，因为在这时调用的是另一个回调 REFLECT_IRET。REFLECT_IRET 清除 VM_Servicing_Int 标志，用以允许 REFLECT_Hw_Int 再次反射中断。

8.4 总　　结

SVCINT 和 REFLECT 说明了 Windows 环境的核心特征是怎样影响 VxD 复位硬件中断的。这两个例子阐述了中断虚拟最重要的细节、怎样和 VxD 通信以及 VxD 怎样与 VMM，VPICD 相互影响。

VxD 经常被用来复位另一种重要类型的中断——软件中断。由于处理软件中断非常简单直接，所以在第 14 章选择处理软件中断作为“多方面的”的技术之一。

本章结束了第二部分的检查 VxD 的基本原理。下一章开始第三部分，提供一些更先进的技术。

VPICD 将硬件中断与虚拟机关联的示意图如图 8-2 所示，从串口 COM1 服务于中断的 VxD 的 SVCINT 见列表 8.1～表 8.3。

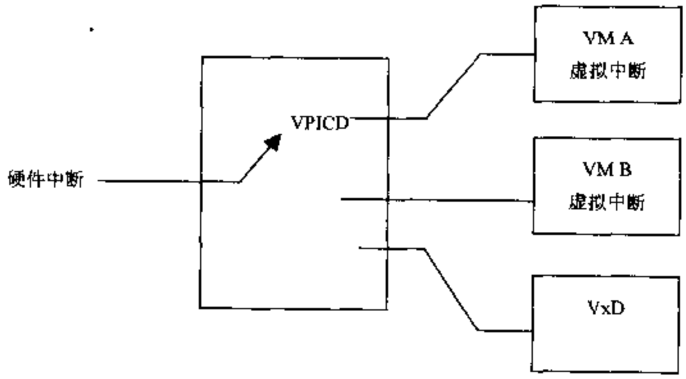

图 8-2　VPICD 将硬件中断与虚拟机关联示意图

列表 8.1　svcint.asm

```
PAGE  58,132
; ================================================
        .386p
; ************************************************
;               INCLUDE
```

```
; ***********************************************
.XLIST
       include   VMM.INC
       include   VPICD.INC
       include   DEBUG.INC
       include   SHELL.INC
.LIST
; ***********************************************
;      VIRTUAL  DEVICE  DECLARATION
; ***********************************************
SVCINT_Device_ID        EQU    1000h
Declare_Virtual_Device SVCINT,1,0,SVCINT_Control,SVCINT_Device_ID,\
             Undefined_Init_Order,SVCINT_API, SVCINT_API
; ***********************************************
; Inittialization  data

VxD_IDATA_SEG

VxD_IDATA_ENDS

; ***********************************************
; Global  data

VxD_DATA_SEG

SVCINT_IRQ_Desc VPICD_IRQ_Descriptor<4,0,OFFSET32 SVCINT_Hw_Int>

SVCINT_IRQ_Handle     dd    0
VM_Owner              dd    0
Device_Name           dd    'S', 'V', 'C', 'I', 'N', 'T',32,32,32
VM_Servicing_Int      dd    0

RXBUFSIZE             EQU   1024
TXBUFSIZE             EQU   512

RxBufIn               dd    OFFSET32 RxBuf
RxBufOut              dd    OFFSET32 RxBuf
RxBuf                 dd    RXBUFSIZE dup('?')
TxBufIn               dd    OFFSET32 TxBuf
TxBufOut              dd    OFFSET32 TxBuf
TxBuf                 dd    TXBUFSIZE dup('?')
```

```
COM1_BASE              EQU   3F8h
COM1_THR               EQU   COM1_BASE+0
COM1_RDR               EQU   COM1_BASE+0
COM1_IER               EQU   COM1_BASE+1
COM1_IIR               EQU   COM1_BASE+2
COM1_LSR               EQU   COM1_BASE+5
COM1_MSR               EQU   COM1_BASE+6

IIR_NONE               EQU   01h
IIR_THRE               EQU   02h
IIR_RXRDY              EQU   04h

IER_THRE               EQU   02h

SVCINT_Api_Table       dd    OFFSET32 SVCINT_Open
                       dd    OFFSET32 SVCINT_Close
                       dd    OFFSET32 SVCINT_Read
                       dd    OFFSET32 SVCINT_Write

SVCINT_API_Open        EQU   0
SVCINT_API_Close       EQU   1
SVCINT_API_Read        EQU   2
SVCINT_API_Write       EQU   3
SVCINT_API_Err_Unknown         EQU   1
SVCINT_API_Err_AlreadyOpen     EQU   2
SVCINT_API_Err_NotOwner        EQU   1

VxD_DATA_ENDS

; *************************************************

VxD_LOCKED_DATA_SEG

VxD_LOCKED_DATA_ENDS

; *************************************************
; Initialization  code

VxD_ICODE_SEG
; *************************************************
;  SVCINT_Device_Init
;  DESCRIPTION:
```

```
; ENTRY:     EBX is SYS VM handle
; EXIT:
; USES:
; ==============================================================

BeginProc  SVCINT_Device_Init
Queue_Out  'SVCINT_Device_Init VM=#ebx',ebx
       ; Install a hareware interrupt handler
       mov        edi,OFFSET32 SVCINT_IRQ_Desc
       VxDcall     VPICD_Virtualize_IRQ
       Jnc        virt_ok
       Debug_Out   'Failed VPICD_Virtualize_IRQ'
       Stc              ; abort VxD load
       Ret

Virt_ok:
       Mov        [SVCINT_IRQ_Handle],eax
       Clc
       Ret
EndProc SVCINT_Device_Init

VxD_ICODE_ENDS

; ***********************************************

VxD_CODE_SEG

; ***********************************************
;  SVCINT_Control
;  DESCRIPTION:dispatch control messages to correct handlers
;  ENTRY:
;  EXIT: Carry clear if no error
;  USES:
; ==============================================================

BeginProc  SVCINT_Control
       Control_Dispatch  Device_Init,SVCINT_Device_Init
       Control_Dispatch  VM_Terminate,SVCINT_VM_Terminate
       Clc
       Ret
EndProc  SVCINT_Control
```

```
BeginProc  SVCINT_API
        Mov        ax, [ebp.Client_AX]
        Cmp        ax,SVCINT_API_Read
        Je         Api_Convert
        Cmp        ax,SVCINT_API_Write
        Je         Api_Convert
        Cmp        ax,SVCINT_API_Open
        Je         Api_Convert
        Cmp        ax,SVCINT_API_Close
        Je         Api_Convert
        Mov        ax,SVCINT_API_Unknown
        Ret

Api_Convert:
        Push        eax
        Mov         ax,(Client_ES SHL 8) or Client_BX
        VMMcall     Map_Flat
        Mov         edi,eax
        Mov         ecx,[ebp.Client_ECX]
        Pop         eax

Api_CheckVM:
        Cmp         [VM_Owner],ebx
        Je          Api_Call
        Mov         [ebp.Client_EAX],SVCINT_API_Err_NotOwner
        Ret

Api_CheckVMNone:
        Cmp         [VM_Owner],0
        Je          Api_Call
        Mov         [ebp.Client_EAX],SVCINT_API_Err_AlreadyOpen
        Ret

Api_Call:
        Movzx       eax,ax
        Shl         eax,2
        Call        SVCINT_API_Table[eax]

        Mov         [ebp.Client_ECX].ecx
        Mov         [ebp.Client_EAX].eax
        Ret
```

```
EndProc  SVCINT_API

BeginProc  SVCINT_VM_Terminate

Queue_Out  'SVCINT_VM_Terminate VM=#ebx',ebx
       Cmp       ebx,[VM_Owner]        ; is VM the device owner
       Jne       vm_not_owner          ; N:nothing to do
       Mov       [VM_Owner],0          ; Y:now device has no owner
vm_not_owner:
       clc
       ret

EndProc  SVCINT_VM_Terminate

; ************************************************
;  SVCINT_Hw_Int
;  DESCRIPTION
;  ENTRY:   EAX=IRQ  handle
          EBX=current VMhandle
;  EXIT:
;  USES:
; ================================================

BeginProc SVCINT_Hw_Int

Queue_Out  'SVCINT_Hw_Init VM=#ebx',ebx
       Cmp       [VM_Owner],0      ; Anybody own the IRQ
       Jne       in_iir                ; Y:service it
       Jmp       Eat_Int               ; N:eat it

In_iir:
       Mov       edx,COM1_IIR
       In        al,dx
       Cmp       al,IIR_NONE
       Je        int_xit

       Cmp       al,_IIR_RXRDY
       Je        int_thre

       Mov       edx,COM1_RDR
       In        al,dx
       Mov       edi,RxBufIn
```

```
        Cmp       edi,OFFSET32 RxBuf+RXBUFSIZE
        Je        norxwrap
        Mov       RxBufIn,OFFSET32 RxBuf
        Mov       edi,RxBufIn
Norxwrap:
        Mov       BYTE PTR[edi],al
        Jmp       SHORT in_iir

Int_thre:
        Cmp       al,IIR_THRE
        Jne       int_xit

        Mov       edi,TxBufOut
        Cmp       edi,TxBufIn
        Jne       txout
        Mov       edx,COM1_IER
        In        al,dx
        And       al,NOT_IRE_THRE
        Jmp       SHORT in_iir

Txout:
        Cmp       edi,OFFSET32 TxBuf+TXBUFSIZE
        Je        notxwrap
        Mov       TxBufIn,OFFSET32 TxBuf
        Mov       edi,TxBufIn
Norxwrap:
        Mov       al, BYTE PTR[edi]
        Mov       edx,COM1_THE
        Out       dx,al
        Jmp       in_iir

Int_xit:
        Mov       eax,[SVCINT_IRQ_Handle]
        VxDcall   VPICD_Phys_EOI
        Clc                          ; int was servied
        Ret

EndProc  SVCINT_Hw_Int

BeginProc  Eat_Int

More_ints:
```

```
        Mov       edx,COM1_IIR
        In        al,dx
        Cmp       al,IIR_NONE          ; interrupt still pending?
        Jne       SHORT ate_it         ; no,do EOI

        ; Brute force approach to clearing UART interrupt:
        ; read all registrs that can clear an interruptting condition
        mov       edx,COM1-MSR
        in        al,dx
        mov       edx,COM1-RDR
        in        al,dx
        mov       edx,COM1-LSR
        in        al,dx
        jmp       SHORT more_ints

ate_it:
        mov       eax,[SVCINT_IRQ_Handle]
        VxDcall    VPICD_Phys_EOI
        Ret

EndProc  Eat_Int

BeginProc  SVCINT_Open

        Mov       [VM_Owner]r,ebx
        Xor       eax,eax
        Ret

EndProc  SVCINT_Open

BeginProc  SVCINT_Close

        Mov       [VM_Owner]r,0
        Xor       eax,eax
        Ret

EndProc  SVCINT_Close

BeginProc  SVCINT_Read

        Push      ecx
        Mov       esi,[RxBufOut]
```

```
Readtop:
        ; ESI→RxBufOut,EDI→caller's buffer
        cmp         esi,[RxBufin]
        je          readdone
        movsb                   ; copy from RxBufOut to caller's buffer
        cmp         esi,OFFSET32 RxBuf+RXBUFSIZE
        jae         readwrap

readloop:
        loop        readtop

readdone:
        mov         [RxBufOut],esi
        pop         eax         ; eax-requested size
        sub         eax,ecx
        mov         ecx,eax     ; ecx-actual size
        xor         eax,eax
        ret

readwrap:
        mov         [RxBufOut],OFFSET32 RxBuf
        jmp         SHORT  readloop

EndProc  SVCINT_Read

BeginProc  SVCINT_Write

        Push        ecx
        Mov         esi,edi
        Mov         edi,[TxBufIn]

Writetop:
        ; ESI→ caller's buffer,EDI→TxBufOut
        cmp         esi,[TxBufOut]
        je          writedone
        movsb                   ; copy from RxBufOut to caller's buffer
        cmp         edi,OFFSET32 TxBuf+TXBUFSIZE
        jae         writewrap

readloop:
        loop        writetop
```

```
writedone:
        mov      [TxBufIn],esi
        pop      eax           ; eax=requested size
        sub      eax,ecx
        mov      ecx,eax        ; ecx=actual size
        xor      eax,eax
        ret

writewrap:
        mov      [TxBufIn],OFFSET32 TxBuf
        jmp      SHORT  writeloop

EndProc  SVCINT_Write

VxD_CODE_ENDS

END
; end of file
```

列表 8.2　svcint.def

```
LIBRARY         SVCINT
DESCRIPTION      'Win386  SVCINT  Device(Version1.0)'
EXETYPE         DEV386
SEGMENTS
                _LTEXT  PRELOAD  NONDISCARDABLE
                _LDATA  PRELOAD  NONDISCARDABLE
                _ITEXT  CLASS  'ICODE'  DISCARDABLE
                _IDATA  CLASS  'ICODE'  DISCARDABLE
                _TEXT   CLASS  'PCODE'  NONDISCARDABLE
                _DATA   CLASS  'PCODE'  NONDISCARDABLE
EXPORTS
                SVCINT_DDB   @1
```

列表 8.3　svcint.mak

```
.all:svcint.386

svcint.obj:svcint.asm
 masm5 -DDEBUG svcint.asm

svcint.386:svcint.obj
 link386 svcint.obj,svcint.386,svcint.map/MAP,,svcint.def;
```

```
  mapsym32 svcint
  addhdr svcint.386
```

列表 8.4 reflect.asm

```
PAGE  58,132
;************************************************
TITLE  vrefd.asm
;************************************************

;=================================================
       .386p
;************************************************
;            INCLUDE
;************************************************
.XLIST
       include   VMM.INC
       include   VPICD.INC
       include   DEBUG.INC
       include   SHELL.INC

.LIST
;************************************************
;     VIRTUAL  DEVICE  DECLARATION
;************************************************
Declare_Virtual_Device REFLECT ,1,0,SVCINT_Control,SVCINT_Device_ID,\
             Undefined_Init_Order
;************************************************
; Inittialization   data

VxD_IDATA_SEG

VxD_IDATA_ENDS

;************************************************
; Global   data

VxD_DATA_SEG

REFLECT_IRQ_Desc VPICD_IRQ_Descriptor<4,0,OFFSET32 REFLECT_Hw_Int,\
             OFFSET32 REFLECT_Virt_Int, OFFSET32 REFLECT_EOI,\
             OFFSET32 REFLECT_Mask, OFFSET32 REFLECT_IRET>
```

```
SVCINT_IRQ_Handle     dd    0
VM_Owner              dd    0
Device_Name           dd    'R', 'E', 'F', 'L', 'E', 'C', 'T',32,32,32
VM_Servicing_Int      dd    0

COM1_BASE           EQU   3F8h
COM1_RBR            EQU   COM1_BASE+0
COM1_IIR            EQU   COM1_BASE+2
COM1_LSR            EQU   COM1_BASE+5
COM1_MSR            EQU   COM1_BASE+6

VxD_DATA_ENDS

VxD_LOCKED_DATA_SEG

VxD_LOCKED_DATA_ENDS

; ***********************************************
; Initialization  code

VxD_ICODE_SEG
; ***********************************************
;  REFLECT_Device_Init
;  DESCRIPTION:
;  ENTRY:     EBX is SYS VM handle
;  EXIT:
;  USES:
; ==============================================

BeginProc  REFLECT_Device_Init

Queue_Out  'REFLECT_Device_Init VM=#ebx',ebx
       ; Install a hareware interrupt handler
       mov        edi,OFFSET32 REFLECT_IRQ_Desc
       VxDcall     VPICD_Virtualize_IRQ
       Jnc        virt_ok
       Debug_Out   'Failed VPICD_Virtualize_IRQ'
       Stc                      ; abort VxD load
       Ret

Virt_ok:
       Mov        [REFLECT_IRQ_Handle],eax
```

```
        Clc
        Ret
EndProc REFLECT_Device_Init

VxD_ICODE_ENDS

; ************************************************

VxD_CODE_SEG

; ************************************************
;  REFLECT_Control
;  DESCRIPTION:dispatch control messages to correct handlers
;  ENTRY:
;  EXIT: Carry clear if no error
;  USES:
; ================================================

BeginProc  REFLECT_Control

        Control_Dispatch  Device_Init,REFLECT_Device_Init
        Control_Dispatch  VM_Terminate,REFLECT_VM_Terminate
        Clc
        Ret

EndProc  REFLECT_Control

BeginProc  REFLECT_VM_Terminate

Queue_Out  'REFLECT_VM_Terminate VM=#ebx',ebx
        Cmp       ebx,[VM_Owner]    ; is VM terminatingthe device owner
        Jne       vm_not_owner      ; N:nothing to do
        Mov       [VM_Owner],0      ; Y:now device has no owner
vm_not_owner:
        clc
        ret

EndProc  REFLECT_VM_Terminate

; ************************************************
;  REFLECT_Hw_Int
;  DESCRIPTION
```

```
; ENTRY:  EAX=IRQ handle
        EBX=current VMhandle
; EXIT:
: USES:
; ==============================================

BeginProc REFLECT_Hw_Int

        Cmp     [VM_Owner],0            ; Anybody own the IRQ
        Je      eat_it                  : no,so eat this interrupt
        Cmp     [VM_Servicing_Int],0    ; Vm of process servicing int?
        Je      eat_it                  ; yes,so eat this interrupt

Eat_it:
        Call    Eat_Int
Int_xit:
        Clc
        Ret

EndProc REFLECT_Hw_Int

BeginProc  Eat_Int

More_ints:
        Mov     edx,COM1_IIR
        In      al,dx
        Test    al,1                    ; interrupt still pending?
        JNe     SHORT ate_it            ; no,do EOI

        ; Brute force approach to clearing UART interrupt:
        : read all registrs that can clear an interruptting condition
        mov     edx,COM1-MSR
        in      al,dx
        mov     edx,COM1-RBR
        in      al,dx
        mov     edx,COM1-LSR
        in      al,dx
        jmp     SHORT more_ints

ate_it:
        mov     eax,[REFLECT_IRQ_Handle]
        VxDcall  VPICD_Phys_EOI
```

```
        Ret

EndProc  Eat_Int

; ************************************************
;  REFLECT_Virt_Int
;  DESCRIPTION:
;  ENTRY:   EAX-IRQ  handle
          EBX=current VMhandle
;  EXIT:
;  USES:
; ================================================

BeginProc REFLECT_Virt_Int

Queue_Out  'REFLECT_Virt_Int VM=#ebx',ebx
        Mov       [VM_Servicing_Int],1
        ret

EndProc REFLECT_Virt_Int

; ************************************************
:  REFLECT_EOI
;  DESCRIPTION:
;  ENTRY:   EAX=IRQ  handle
          EBX-current VMhandle
;  EXIT:
;  USES:
; ================================================

BeginProc REFLECT_EOI

Queue_Out  'REFLECT_EOI VM-#ebx',ebx
        VxDcall    VPICD_Phys_EOI

        Mov       ebx, [VM_Owner]
        VxDcall    VPICD_Clear_Int_Request
        ret

EndProc REFLECT_EOI

; ************************************************
```

```
; REFLECT_Mask
; DESCRIPTION:
; ENTRY:  EAX=IRQ handle
        EBX=current Vmhandle
        ECX=0 if unmasking,non-zero if masking
; EXIT:
; USES:
; ================================================

BeginProc REFLECT_Mask

       Jecxz      SHORT unmask
Queue_Out  'REFLECT_Mask VM=#ebx',ebx
       Mov        [VM_Owner],0
       VxDcall    VPICD_Physically_Mask
       ret

Unmask:
Queue_Out  'REFLECT_Mask(unmask) VM-#ebx',ebx
       Cmp        [VM_Owner],0          ; does device have an owner?
       Je         new_owner             ; no,then assign new one

       Mov        eax,[VM_Owner]        ; EAX=VM owner
       Lea        esi,[Device_Name]     ; ESI=device name

       VxDcall    SHELL_Resolve_Contention
       Mov        ebx,eax           ; EBX=new VM owner

New_owner:
Queue_Out  'Assigning owner VM=#ebx',ebx
       Mov        [VM_Owner],ebx
       Mov        eax,[REFLECT_IRQ_Handle]
       VxDcall    VPICD_Physically_Unmask
       ret

EndProc REFLECT_Mask

; ***************************************************
: REFLECT_IRET
; DESCRIPTION:
; ENTRY:  EAX-IRQ handle
        EBX=current VMhandle
```

```
; EXIT:
; USES:
; ==============================================

BeginProc REFLECT_IRET

Queue_Out  'REFLECT_IRET VM=#ebx',ebx
       Mov        [VM_Servicing_Int],0
       ret

EndProc REFLECT_IRET

VxD_CODE_ENDS

END
; End of file
```

列表 8.5 reflect.def

```
LIBRARY        REFLECT
DESCRIPTION     'Win386  REFLECT  Device(Version1.0) '
EXETYPE        DEV386
SEGMENTS
              _LTEXT  PRELOAD  NONDISCARDABLE
              _LDATA  PRELOAD  NONDISCARDABLE
              _ITEXT  CLASS  'ICODE'  DISCARDABLE
              _IDATA  CLASS  'ICODE'  DISCARDABLE
              _TEXT   CLASS  'PCODE'  NONDISCARDABLE
              _DATA   CLASS  'PCODE'  NONDISCARDABLE
EXPORTS
              REFLECT_DDB   @1
```

列表 8.6 reflect.mak

```
.all: reflect.386

reflect.obj:reflect.asm
  masm5 -DDEBUG reflect.asm

reflect.386:reflect.obj
  link386 reflect.obj,reflect.386,reflect.map/MAP,,reflect.def;
  mapsym32 reflect
  addhdr reflect.386
```

第三部分

特殊技术

第 9 章　第 0 层处理器缩短响应时间

写中断作为中断驱动服务子程序时，基于 DLL 的 Windows 设备驱动程序会需要非常长的响应时间。Windows 的 ISR 比 DOS 的 ISR 占用的运行时间长，因为 Windows 的 ISR 直到 Windows 调度了相应的 VM 才能运行。如果将 ISR 放入 VxD 中，就解决了这个问题，但是需要修改用户的 SYSTEM.INI 文件（以便 Windows 在初始化时加载 VxD）。本章将介绍另外一种技术，它把这个调度在 ISR 的运行等待时间提前安排移到上层的替换。即使一个 DLL 仍然在 DLL 中，它也被安装在 Windows 管理程序的环境中运行，而非系统 VM 的环境中。这种由 VM 安装在管理程序中运行的 ISR 称为第 0 层处理器。

9.1　替换 VxD

如果没有 VxD 为中断注册，VPICD 通常将中断反射为相应的 VM，VM 可以通过调度机制的“退出”来避免反射过程。如果反射过程通过 VPICD 注册了一个第 0 层的处理器，那么 VPICD 就只将中断调度到相应第 0 层的处理器。服务 VPICD_Install_Handler () 是 VxD 初始化中断的注册服务的一种特殊形式，它提供一个特殊的转换服务，允许你不用支持整个 VxD 接口而创建第 0 层处理器。这个方案在图 9-1 中作了说明。

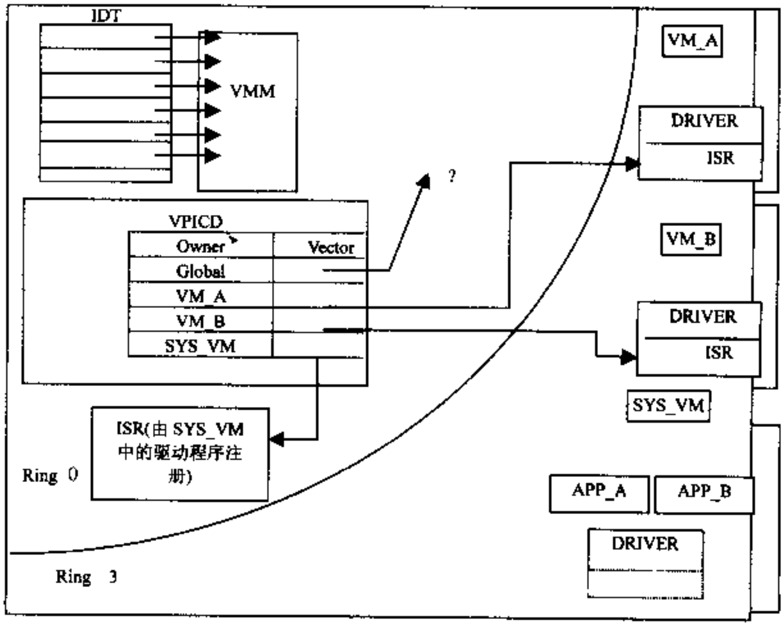

图 9-1　VPICD_Install_Handler () 创建第 0 层处理器分析图

从原理上讲，这个方法很简单，而且与作为 VxD 的一部分安装处理器非常类似。完成的结果却不简单。记住，你写入的是一个放入 VM’S 环境的 16 位中断处理器，它将在第 0 层的 32 位环境中执行。幸运的是，Windows 通过对内存访问的自动转换来将这个不匹配中的许多细节隐藏起来。

9.1.1 安装

安装第 0 层处理器比安装一般的处理器涉及到的东西多。要安装处理器，必须有以下几个步骤：

1. 分配和初始化 BIMODAL_INT_STRUC 结构，用来描述注册的处理器和它将要服务的中断。
2. 获得 VPICD 的每个入口点。
3. 调用 VPICD_Install_Handler（）服务注册处理器并将它传给 BIMODAL_INT_STRUC。

9.1.2 初始化 BIMODAL_INT_STRUC

由于处理器和 VPICD 都在中断时访问 BIMODAL_INT_STRUC，所以这个结构必须像其他中断安全缓冲区一样被分配，它必须在固定的和页面锁定的段中。另外，BIMODAL_INT_STRUC 被分配到与驱动程序环境结构相同的段非常重要。当处理器在第 0 层被调用时，由 VMM 提供的唯一环境是一个指向 BIMODAL_INT_STRUC 的指针。在与 BIMODAL_INT_STRUC 相同的段里分配设备文本，使得查找设备文本给于 BIMODAL_INT_STRUC 的指针更加容易。

图 9-2 说明了结构的一些字段由驱动程序 DLL 初始化，而其他一些字段在注册期间被 VPICD 初始化。首先介绍那些在调用 VPICD 服务前被第 0 层的 DLL 初始化的字段。本书将在后面的内容中介绍由 VPICD 服务填写的字段。

图 9-3 给出了对结构的 C 语言描述。第一个字段 BIH_Irq_Number 是中断号，处理器必须与之相对应。它是中断号，而不是向量，所以它的值从 0 到 15。如果这个 IRQ 已经被其他处理器注册为非共享中断，那么调用 VPICD_Install_Handler 就会失败。

BIH_Entry 字段是一个指向中断处理器的 NEAR 指针。

字段 BIH_User_Mode_CS 和 BIH_User_Mode_DS 必须与中断处理器程序和数据段一起初始化。最后一个 BIMODAL_INT_STRUC 的字段是数组 EBIS_SEL_STRUS。EBIS_SEL_STRUS 的 C 语言描述如下：

```
typedef struct{
  USHORT  EBIS_User_Mode_Sel;
  USHORT  EBIS_User_Mode_Off;
  USHORT  EBIS_Super_Mode_Sel;
} EBIS_SEL_STRUS;
```

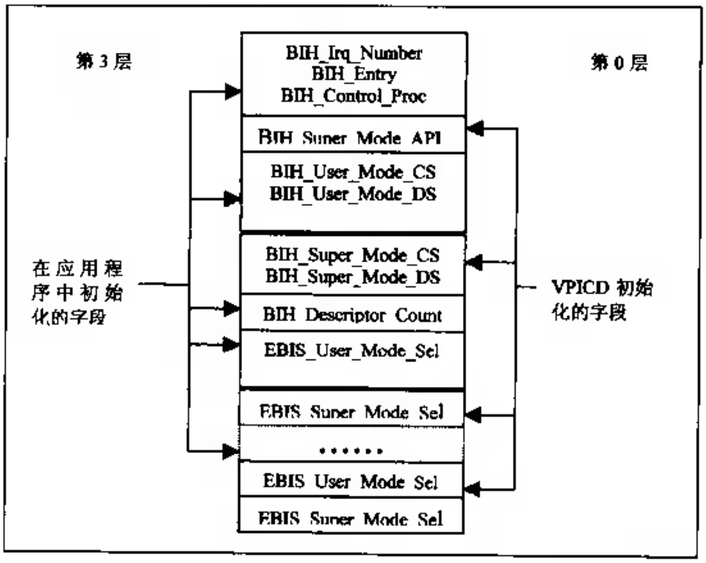

图 9-2　结构的一些字段由驱动程序 DLL 初始化

```
Typedef  struct
{
  WORD     BIH_Irq_Number;
  WORD     BIH_VM_ID;
  DWORD    BIH_Next;
  DWORD    BIH_Reserved1;
  DWORD    BIH_Reserved2;
  DWORD    BIH_Reserved3;
  DWORD    BIH_Reserved4;
  DWORD    BIH_Flags;
  WORD     BIH_Mode;
  WORD     BIH_Entry;
  WORD     BIH_Control_Proc;
  WORD     BIH_Pad;
  DWORD    BIH_User_Mode_API;
  DWORD    BIH_Super_Mode_API;
  WORD     BIH_User_Mode_CS;
  WORD     BIH_User_Mode_DS;
  WORD     BIH_Super_Mode_CS;
  WORD     BIH_Super_Mode_DS;
  WORD     BIH_DescrCount;
  EBIS_SEL_STRUC      BIH_Descriptor[2];
}BIMODAL_INT_STRUC;
```

图 9-3　Bimodal_Int_Struc 结构的语言描述

当驱动程序需要访问它自己的数据段之外的数据时使用这个结构。驱动程序使用两个动态分配的缓冲区，这两个动态分配的缓冲区是全局性的（GlobalAlloced），因此置于它自己的段之外，使用两个 EBIS_SEL_STRUS 作为结构最后的字段。

例子驱动程序用传送层缓冲区的向量初始化 EBIS_SEL_STRUC 的第一个 EBIS_User驶_Mode_Sel 字段，并将字段 BIH_Descriptor_Count 的值设为 2（段的数目）。第二个 EBIS_User_Mode_Sel 字段为接收层缓冲区初始化。如果处理器不需要访问它自身之外的数据段，设计的结构就不需要字段 EBIS_User_Mode_Sel，而且还必须将 BIH_Descriptor_Count 的值设为 0。

许多剩余的字段是被保留的。字段 BIH_Reserved1 到 BIH_Reserved4 明显属于这一类，字段 BIH_Next 也是。字段 BIH_Flags 不能使用，必须由你的 DLL 设为 0。字段 BIH_Control_Proc 也不能使用，但是它必须指向一个伪过程，进行 FAR SET。

9.2　注册处理器

例子驱动程序将所有需要的安装处理器的程序代码压缩到 Install_Ring 0_Handler 函数中（如图 9-4）。这个函数找到 VPICD 的入口点，填充结构，并且调用 VPICD 的安装服务。

```
BOOL Install_Ring0_Handler(DEVICEINFO *hPort)
{
  WORD   sel;
  RING0  *pRing0;
  WORD   myds;

asm
{
xor    di,di
mov    es,di
mov    ax,0x1684
mov    bx,id_vpicd
int    0x2f
mov    WORD PTR [ptn_VPICD_Entry],di
mov    WORD PTR [ptn_VPICD_Entry+2],es
mov    ax,es
or     ax,di
jz     xit1

mov    ax,VPICD_Get_Version
call   [pfn_VPICD_Entry]
cmp    ax,0x30A
```

```
iea    xit1
mov    WORD PTR [ptn_VPICD_Entry],0
mov    WORD PTR [ptn_VPICD_Entry+2],0
xit1:
  }
if(pfn_VPICD_Entry==NULL)
return(FALSE);
  pRing0=&(hPort→Ring0Info);
  memset(pRing0,0,sizeof(RING0));
  pRing0→IrqNumber=hPort→Irq;
  pRing0→IsrProc=OFFSET(Asm_Ring0_Isr);
  pRing0→ControlProc=OFFSET(Asm_Ring0_Control);
pRing0→Ring3Cs=SELECTOROF(Asm_Ring0_Isr);
  PRing0→Ring3DS=myds;
  pRing0→DescrCount=1;
  sel=SELECTOROF(hPort→TxBuf.Start);
  pRing0→Descriptor[0].Ring3Sel=sel;

_asm
{
mov    ax,VPICD_Install_Handler
mov    di,WORD PTR pRing0
push   ds
pop    es
call   [pfn_VPICD_Entry]
jnc    xit2
xor    ax,ax
xit2:
  }
  return(TRUE)
}
```

图 9-4　在函数 Install_Ring0_Handler 中安装处理器所需的代码

驱动程序必须首先找到 VPICD 入口点的地址。使用编译器的嵌套汇编语言特性，Install_Ring0_Handler 发布 INT 2FH，其中 AX=1684H(获取设备 API 入口点的函数)，BX=3H（VPICD 的虚拟设备 ID 号）。入口点地址在 ES：DI 中返回。如果 ES：DI 等于 0，说明设备不存在，它没有 API 入口点，或者 Windows 不是增强模式。对于以上的每种情况，函数返回 FALSE。

如果 ES：DI 不是 0，Install_Ring0_Handler 就通过调用 ES：DI 中 VPICD 的入口点注册函数来指向驱动程序初始化的 BIMODAL_INT_STRUC，以及在 AX 中的 Install_Ring0_Handler 的函数代码（2H）。如果返回清除 Carry 标志，则注册成功，并且初始化函数返

回 TRUE；否则，函数通过返回 FALSE 的形式报错。

9.2.1　VPICD 初始化字段

作为注册过程的一部分，VPICD 在 BIMODAL_INT_STRUC 中初始化几个字段。首先，VPICD 为每个处理器的入口点创建一个“GDT 别名”，并且将这个别名存储到 BIMODAL_INT_STRUC 的 BIH_Super_Mode_CS 字段中。

这个别名由驱动程序的存储字段 BIH_User_Mode_CS 和 BIH_Entry 构成，同时使用这两个字段来生成一个指向处理器的指针。但是有两个原因使 VPICD 不能使用那个地址来调用处理器。第一，在 BIH_User_Mode_CS 中的选择器是第 3 层的优先级，VPICD 运行在第 0 层的 VxD。如果 VPICD 通过这个第 3 层的选择器来调用第 0 层的处理器，微处理器必须进行非常复杂的层转换。第二，BIH_User_Mode_CS 是一个 LDT 选择器，而不是 GDT 选择器。每个 VM 都有自己的 LDT。如果中断发生时另一个 VM 正在执行，那么它的 LDT 就可能被使用，而第 0 层处理器的选择器没有被使用，因为它属于不同的 LDT。

GDT 选择器就没有这个问题，因为只有一个 GDT，所以 VPICD 为 BIH_User_Mode_CS 选择器创建一个 GDT 别名。别名选择器有和原来一样的基地址和界限，但是它位于 GDT 中。VPICD 给了这个别名一个第 0 层的优先级，而原始的 BIH_User_Mode_CS 是第 3 层优先级。

当中断发生时，VPICD 用 BIH_Super_Mode_CS 中的选择器和 BIH_Entry 中的偏移量形成一个远程指针。接着 VPICD 使用这个远程指针调用驱动程序 DLL 中的处理器。由于 BIH_Super_Mode_CS 中的选择器是第 0 层的优先级，所以在 VPICD 调用处理器时不需要进行层转换。

VPICD 还为 BIH_User_Mode_CS 创建一个 GDT 别名，并将这个别名存到 BIH_Super_Mode_CS 中。虽然 BIH_Super_Mode_CS 得到一个第 0 层的优先级，但是 BIH_Super_Mode_CS 中的数据选择器只有第 3 层的优先级。

使用这两个别名，VPICD 可以在第 0 层调用处理器，处理器就使用 BIH_Super_Mode_CS 中的选择器来访问它的数据段。

为了构成每个处理器入口点的别名，VPICD 还将为那些在 BIH_Descriptor_Count 中申请的数据段构成别名。VPICD 查看每个 EBIS_SEL_STRUC 中的 EBIS_User_Mode_Sel 字段，为它们都创建一个 GDT 别名，并将这个别名选择器存储到 EBIS_SEL_STRUC 的 EBIS_Super_Mode_Sel 字段中。于是第 3 层的数据选择器就可以被传送到应用程序或者 DLL。

最后，VPIVD 在 BIH_Super_Mode_API 字段中存储 VPICD 入口点的地址。第 0 层的处理器使用这个入口点在中断时调用 VPICD 的服务（通过 AX 寄存器确定使用哪个服务）。

剩下的两个字段（BIS_Mode 和 BIS_User_Mode_API）对于执行第 0 层的处理器不是必须的，但是如果想将本章的第 0 层的驱动程序与第 10 章的双模态驱动程序相结合，就要使用它们。Windows 中的串口驱动程序 COMM.DRV 就是起这种作用，使一个单一的中断处理器变成在标准模式和增强模式下都有效的中断服务。

9.2.2　处理中断

第 0 层的 ISR 处理中断与传统的处理中断类似。不同之处在于调用 PostMessage（），产生一个 EOI，以及将在 9.2.3 节讲述的编码问题。

这个中断处理器加了一位原来没有的功能位。如果接收层缓冲区溢出，第 0 层的 ISR 就发出一个用户定义的消息，使用 Windows 的控制在 DeviceOpen（）传送。这说明了从第 0 层调用第 3 层代码 PostMessage（）的技术。

有两个原因使 ISR 不能直接调用 PostMessage（）。一个较重要的是 ISR 在第 0 层，而 PostMessage（）在第 3 层。另一个是在 ISR 执行时 System VM 可能不是当前的 VM。所以 ISR 必须将对 PostMessage（）的调用延迟到 SystemVM 执行为止，并且还必须进行第 0 层到第 3 层的适当转换。

VPICD 提供了另一个只需要简单调用就能解决这些问题的服务。第 0 层的 ISR 使用 VPICD 的 BIH_API_CALL_BACK 服务来注册一个回调函数，这个回调函数将在 System VM 环境的第 3 层被调用，是回调函数产生 PostMessage（），而不是 ISR。

在返回时，中断处理器通知中断控制器中断一直被执行，这个通知通过向控制器发送一个 EOI 命令来执行。在接到 EOI 之前，控制器将忽略这个中断级别的其他中断（以及更低优先级的中断），允许中断处理器打开微处理器的中断而不用立即再进入。

Windows 不遵循这个通常的顺序。相反，VPICD 总是立即屏蔽（关）中断级别，并且产生 EOI。被屏蔽的中断级别防止处理器被再进入，但 EOI 允许控制器从其他设备识别中断。

VPICD 最后调用为这个特殊设备安装的“实”中断处理器。如果处理器在 Windows 或 DOS 应用程序中，处理器就发送一个 EOI。VPICD 在试图访问特权端口时捕获 EOI，并且 VPICD 不屏蔽（开）中断级别。

当然，EOI 的捕获不用依靠第 0 层处理器的执行，因为它运行在最高的优先级别上。为了防止 VPICD 和硬件 PIC 失步，处理器必须遵守同样的协定，代替发生 EOI。第 0 层的处理器必须通过 BIH_SUPER_MODE_API 入口点调用 VPICD 的 EOI 服务，其中 AX＝BIH_PAI_EOI（0H）。接着 VPICD 在控制器中不屏蔽（开）中断级别，而且控制器将在那个级别识别中断。

第 0 层的处理器还执行一些与一般处理器不同的东西。在调用 VPICD 的 EOI 服务后，处理器要清除 carry 标记（指示中断被服务了），并返回到 VPICD 的一个 PAR RET。

9.3　编写处理器程序

第 0 层的处理器被编写为两部分：一个小的汇编语言“捕获器”（见列表 9.5）用来存储和恢复寄存器，以及建立数据段寄存器和一个完成相同任务的 C 函数（见列表 9.7）。C 函数这部分包括嵌入的用来调用 VPICD 的 EOI 服务汇编语言，并在返回时设置 carry 标记。

大部分中断处理器可以通过使用关键字 interrupt 用 C 语言编写。这个关键字告诉编译器产生存储寄存器的代码，并且用 DGROUP 加载 DS。由于第 0 层的处理器需要 DS 加载一个第 0 层的选择器，而不是 DGROUP，所以加载正确的选择器是汇编语言的任务。

在将控制权传送给第 0 层的 ISR 前，VPICD 将一个选择器和偏移量调入到 ES：DI，指向与调用 VPICD_Install_Handler（）中使用相同的 BIMODAL_INT_STRUC。由于 BIMODAL_ INT_STRUC 位于处理器的数据段（在 DEVICECONTEXT 之后），所以处理器可以通过拷贝 ES 到 DS 来获得正确的选择器。

ISR 的 C 语言部分还需要一个指向 DEVICECONTEXT 的指针（这个结构包括控制器所需要的关于中断设备的所有内容：它的 I/O 地址、与端口相关的缓冲区和一些状态信息）。可以通过从 SI 中的偏移量减去 DEVICECONTEXT 结构的大小来为这个指针建立一个适当的偏移量。但是不幸的是，不能将这个值传到堆栈去，所以通过 SI 寄存器来传递它。所有这些只需要 3 行汇编语言就能实现：

```
Mov   si,es
Mov   ds,si
Lea   si,[di-SIZE DeviceContext]
```

C 语言模块使用下面的办法来将偏移量拷贝到一个全局指针：

```
_asm mov  hPort,si
```

由于有这些汇编语言的准备，程序其他部分可以是一般的 C 语言，但是有一个重要的原则：第 0 层 ISR 中的 C 程序不能使用堆栈访问。

9.3.1　为什么没有堆栈变量

现在 Windows 应用程序的编译器通过一个 16 位堆栈产生代码。由于处理器运行在第 0 层，这个堆栈的选择器必须有第 0 层的权限，否则对堆栈的访问将导致一个异常事件。不幸的是，VPICD_Install_Handler（）服务不能提供这个组合。下面来看看其原因。

假设处理器中包括图 9-5 给出的代码，编译这些代码将产生图 9-6 所示的汇编语言。由于 VPICD 是 VxD，所以它与代码、数据和堆栈一样，使用 32 位的段。另一方面，第 0 层的处理器是 DLL 的一部分，所以和其他 DLL 一样使用 16 位的代码段。当处理器（16 位）试图调用堆栈时（VPICD 是 32 位的堆栈），它将不会产生动作。

冲突是由微处理器在计算有效地址时使用的选择基址寄存器的算法产生的。访问隐含堆栈的指令如 PUSH 和 POP，使用堆栈段的大小（16 位或 32 位）来选择 SP 或者 ESP；访问显示堆栈的指令如 CMP [BP+6]，使用代码段的大小来选择 BP 或 EBP。所以，当 foo（）中的指令 PUSH 将参数 1 压入堆栈时，这个参数就进入到 32 位 ESP 确定的位置。QueryDevice（）使用 CMP 访问同一个参数，它使用 16 位的 BP 寄存器计算地址，结果是一个无效的访问。

```
void foo(void)
{
  QueryDevice(1);
}
void QueryDevice(int status)
{
  if(status==1)
    //do something
  else
    //do something else
}
```

图 9-5　第 0 层处理器的简单代码

```
_foo:
MOV    BX,1
PUSH   BX
CALL   _QUERYdEVICE
ADD    SP,2
RET

_QUERYdEVICE
PUSH   BP
MOV    BP,SP
CMP    [BP-4],1
```

图 9-6　对应图 9-5 中的代码的汇编语言

9.3.2　一种更糟的情况

当遇到问题时，处理器会切换到它自己的 16 位堆栈，堆栈将处理器修改到 GlobalAlloc（GMEM_FIXED）。为了获得堆栈的 GDT 别名，还要扩展 BIMODAL_INT_STRUC 结构，使它包括一个额外的 EBIS_Sel_Struc。需要将这个新的 EBIS_Sel_Struc 初始化为新的堆栈段，接着运行新的具有 16 位的堆栈处理器，结果在它调入 SS 时产生一个 GP 错误。

使用调试器（WDEB386）清除 GDT，并检测调入 SS 的选择器。选择器（由 VPICD_Install_Handler 返回）有的是第 3 级别的优先级，16 位代码使用的是 16 位的堆栈，但它不是第 0 层的堆栈。计算机的保护模式要求当运行指令 MOV　SS 时，调入选择器的优先级必须等于代码段的优先级。16 位代码运行在第 0 级，但是 VPICD 只给一个第 3 层优先级的选择器，这就导致了 GP 错误的发生。

底行是一个第 0 层处理器，它需要第 0 层 16 位堆栈来正确运行，服务 VPICD_Install_Handler（）不能提供。即使是 DPMI 也不能提供帮助，因为 DPMI 的选择器函数只工作

在 LDT 选择器上，而不是 GDT 选择器。根据经验所知，除了写一个 VxD 外，没有其他办法来分配一个合适的堆栈段。解决的方案就是用 C 语言来写处理器，但是要使用全局变量代替堆栈的参数。

9.4 总　　结

如果用 C 语言（或者汇编语言）编写程序时有一些编程限制，就需要用到第 0 层处理器的一些重要特性。反应事件比使用一般的处理器要好得多，因为第 0 层处理器不需要等到中断反射的条件正确后再执行。由于处理器在第 0 层直接调用，它还可以避免层转换（可能需要上百个时钟周期）所付出的代价。

列表 9.1　ring0.h（0 层处理器）

```
#include <windows.h>

#define  FLAGS_ON_SLAVE_PIC    0x01
#define  FLAGS_OPNE            0x02
#define  FLAGS_RXQOVER         0x04

#define  MASTER_PIC_CTRL       0x20
#define  MASTER_PIC_MASK       0x21
#define  SLAVE_PIC_CTRL        0xA0
#define  SLAVE_PIC_MASK        0xA1
#define  EOI                   0x20

#define SET(value,mask)        value |=mask
#define CLR(value,mask)        value &=(~mask)

typedef  struct
{
   WORD  usReadBufSize;
}DRIVERCAPS, FAR *PDRIVERCAPS;
typedef  PDRIVERCAPS FAR * PDRIVERCAPS

typedef void(FAR interrupt *VOIDINTPROC)();
typedef void(FAR *VOIDINTPROC)()

typedef struct
{
   char  far *Start;
WORD  Size;
```

```
WORD  In;
WORD  Out;
}BUFINFO;

#pragma pack(1)
typedef struct
{
WORD  EBIS_User_Mode_Sel;
WORD  EBIS_User_Mode_Off;
WORD  EBIS_Super_Mode_Sel;
}EBIS_SEL_STRUC
#pragma pack()

#pragma pack(1)
typedef struct
{
WORD            BIH_Irq_Number;
WORD            BIH_VM_ID;
DWORD           BIH_Next;
DWORD           BIH_Reserved1;
DWORD           BIH_Reserved2;
DWORD           BIH_Reserved3;
DWORD           BIH_Reserved4;
DWORD           BIH_Flags;
WORD            BIH_Mode;
WORD            BIH_Entry;
WORD            BIH_Control_Proc;
WORD            BIH_Pad;
DWORD           BIH_User_Mode_API;
DWORD           BIH_Super_Mode_API;
WORD            BIH_User_Mode_CS;
WORD            BIH_User_Mode_DS;
WORD            BIH_Super_Mode_CS;
WORD            BIH_Super_Mode_DS;
WORD            BIH_DescrCount;
EBIS_SEL_STRUC  BIH_Descriptor[2];
}BIMODAL_INT_STRUC;
#pragma pack()

#pragma pack(1)
typedef struct
{
```

```
WORD                usDevNumber;
WORD                usIoBase;
BYTE                bIrq;
BYTE                bFlags;
HWND                hwnd;
VOIDINTPROC        pfOldHandler;
BUFINFO            RxBuf;
BUFINFO            TxBuf;
DRIVERPARAM        params;
VOID                numints;
//MUST be last field in DEVICEINFO
BIMODAL_INT_STRUCT   Ring0Info;
}DEVICECONTEXT, FAR *HDEVICE;
#pragma pack()

HDEVICE FAR PASCAL DeviceOpen(HWND hwnd);
int FAR PASCAL DeviceClose(HDEVICE);
int FAR PASCAL DeviceGetReadStatus(HDEVICE, LPWORD pusStatus);
int FAR PASCAL DeviceGetWriteStatus(HDEVICE, LPWORD pusStatus);
int FAR PASCAL DeviceWrite(HDEVICE, LPBYTE lpData, LPWORD pcBytes);
int FAR PASCAL DeviceRead(HDEVICE, LPBYTE lpData, LPWORD pcBytes);
int FAR PASCAL DeviceSetDriverParams(HDEVICE, PDRIVERPARAMS pParams);
int FAR PASCAL DeviceGetDriverParams(HDEVICE, PDRIVERPARAMS pParams);
int FAR PASCAL DeviceSetDriverCapabilities(HDEVICE, PPDRIVERCAPS
ppDriverCaps);

/*End of file.*/
```

列表 9.2 ring0.c(C 语言 0 层处理器)

```
#inclde <dos.h>
#inclde <conio.h>
#inclde <windows.h>
#inclde <windowsx.h>
#inclde "ring0.h"
#inclde "uart.h"
#inclde "ring0asm.h"

#define VPICD_ID                 0x03
#define VPICD_GetVersion         0x00
#define VPICD_Install_Handler    0x01
#define VPICD_Remove_Handler     0x02
#define VPICD_Call_At_Ring0      0x03
```

```
DEVICECONTEXT  Device1(0,0x3F8,4,0,NULL);
DRIVERPARAMS   DefaultParams(16);
DRIVERPARAMS   DriverCaps(0x0101);
VOIDFARPROC    pfVPICDEntry;

BOOL ValidHandle(HDEVICE hDevice);
BOOL Install_Ring0_Handler(DEVICECONTEXT *hDevice);
VOID Remove_Ring0_Handler(DEVICECONTEXT *hDevice);

HDEVICE FAR PASCAL DeviceOpen(HWND hwnd)
{
HDEVICE  hDevice;
BYTE     mask;

OutputDebugString("DeviceOpen\n");
HDevice=&Device1;
if(hDevice→bFlags&FLAGS_OPEN)
      return (HDEVICE)-1;
hDevice→params=DefaultParams;

  hDevice→RxBuf.Size= hDevice→params.usReadBufSize;
hDevice→RxBuf.Start=GlobalAllocPtr(GMEM_FIXED|GMEM_NODISCARD,
                          hDevice→RxBuf.Size);
if!(hDevice→RxBuf.Start)
{
   OutputDebugString("ERROR\n");
   Return (HDEVICE)-1;
}
hDevice→RxBuf.In= hDevice→RxBuf.Out=0;

  hDevice→TxBuf.Size= hDevice→params.usReadBufSize;
hDevice→TxBuf.Start=GlobalAllocPtr(GMEM_FIXED|GMEM_NODISCARD,
                          hDevice→TxBuf.Size);
if(!hDevice→TxBuf.Start)
{
   OutputDebugString("ERROR\n");
   Return (HDEVICE)-1;
}
hDevice→TxBuf.In= hDevice→TxBuf.Out=0;
hDevice→hwnd=hwnd;
```

```
    if(hDevice->bIrq<8)
    {
    mask=_inp(MASTER_PIC_MASK);
    SET(mask,1<<hDevice->bIrq);
    _outp(MASTER_PIC_MASK,mask);
    }
else
{
    SET(hDevice->bFlags,FLAGS_ON_SLAVE_PIC);
    mask=_inp(SLAVE_PIC_MASK);
    SET(mask,1<<hDevice->bIrq-8);
    _outp(SLAVE_PIC_MASK,mask);
    }

if(!Install_Ring0_Handler((DEVICECONTEXT *)hDevice))
    return (HDEVICE)-1;

//Configure UART
_outp(hDevice->usIoBase+UART_REG_IER,0);
_outp(hDevice->usIoBase+UART_REG_LCR,UART_LCR_DLAB);
_outp(hDevice->usIoBase+UART_REG_BAUDLO,BAUD_1200);
_outp(hDevice->usIoBase+UART_REG_BAUDNI,0);
_outp(hDevice->usIoBase+UART_REG_LCR,UART_LCR_8N1);
_outp(hDevice->usIoBase+UART_REG_IER,UART_IER_RXRDY);
_outp(hDevice->usIoBase+UART_REG_MCR,UART_MCR_OUT2);

//Unmask interrupt at pic
if(hDevice->bIrq<8)
{
    mask=_inp(MASTER_PIC_MASK);
    CLR(mask,1<<hDevice->bIrq);
    _outp(MASTER_PIC_MASK,mask);
}
else
{
    mask=_inp(SLAVE_PIC_MASK);
    CLR(mask,1<<hDevice->bIrq-8);
    _outp(SLAVE_PIC_MASK,mask);
}

SET(hDevice->bFlags,FLAGSOPEN);
return hDevice;
```

```
}

int FAR PASCAL _export DeviceClose(HDEVICE hDevice)
{
BYTE mask;
OutputDebugString("DeviceClose\n");
if(!ValidHandle(hDevice))
   return -1;
if((hDevice->bFlags & FLAGS_OPEN)==0)
   return FALSE;
CLR(hDevice->bFlags,FLAGS_OPEN);

//disable UART interrupt
_outp(hDevice->usIoBase+UART_REG_IER,0);
_outp(hDevice->usIoBase+UART_REG_MCR,0);

if(hDevice->bIrq<8)
{
   mask=_inp(MASTER_PIC_MASK);
   SET(mask,1<<hDevice->bIrq);
   _outp(MASTER_PIC_MASK,mask);
}
else
{
   SET(hDevice->bFlags,FLAGS_ON_SLAVE_PIC);
   mask=_inp(SLAVE_PIC_MASK);
   SET(mask,1<<hDevice->bIrq);
   _outp(SLAVE_PIC_MASK,mask);
}
Remove_Ring0_Handler((DEVICECONTEXT *)hDevice);
GlobalFreePtr(hDevice->RxBuf.Start);
GlobalFreePtr(hDevice->TxBuf.Start);

return 0;
}

int FAR PASCAL DeviceGetReadStatus(HDEVICE hDevice,LPWORD pusStatus)
{
OutputDebugString("DeviceGetReadStatus\n");
if(!ValidHandle(hDevice))
   return -1;
if((hDevice->bFlags & FLAGS_OPEN)==0)
```

```
    return -1;

if(hDevice→RxBuf.In!=hDevice→RxBuf.Out)
    *pusStatus=1;
else
    *pusStatus=0;

return 0;
}

int FAR PASCAL DeviceGetWriteStatus(HDEVICE hDevice,LPWORD pusStatus)
{
OutputDebugString("DeviceGetWriteStatus\n");
if(!ValidHandle(hDevice))
    return -1;
if((hDevice→bFlags & FLAGS_OPEN)==0)
    return -1;

if(_inp(hDevice→usIoBase+UART_REG_LSR) & UART_LSR_THRE)
    *pusStatus=1;
else
    *pusStatus=0;

return 0;
}

int FAR PASCAL DeviceWrite(HDEVICE hDevice,LPBYTE lpData,LPWORD pcBytes)
{
WORD i;
BYTE ier;

OutputDebugString("DeviceWrite\n");
if(!lpData)
    return -1;
if(!ValidHandle(hDevice))
    return -1;
if((hDevice→bFlags & FLAGS_OPEN)==0)
    return -1;

for(i=0; i<*pcBytes; i++)
{
    hDevice→TxBuf.Start[hDevice→TxBuf.In++]=lpData[i];
```

```
    if(hDevice→TxBuf.In>=hDevice→TxBuf.Size)
        hDevice→TxBuf.In=0;
}

if(UART_LSR_THRE & _inp(hDevice→usIOBase+UART_REG_LSR))
{
    ier=_inp(hDevice→usIoBase+UART_REG_LER);
    if((UART_IER_THRE & ier)==0)
    {
        _outp(hDevice→usIoBase+UART_REG_LER,ier| UART_IER_THRE);
    }
}

return 0;
}

int FAR PASCAL DeviceRead(HDEVICE hDevice,LPBYTE lpData,LPWORD pcBytes)
{
WORD i,cBytes;

OutputDebugString("DeviceRead\n");
if(!lpData)
    return -1;
if(!ValidHandle(hDevice))
    return -1;
if((hDevice→bFlags & FLAGS_OPEN)==0)
    return -1;

cBytes=*pcBytes;
for(i=0; i<cBytes; i++)
{
    if(hDevice→RxBuf.In>=hDevice→RxBuf.Out)
        break;
    lpData[i]=hDevice→RxBuf.Start[hDevice→RxBuf.Out++];
    if(hDevice→RxBuf.Out>=hDevice→RxBuf.Size)
        hDevice→RxBuf.Out=0;
    *pcBytes--;
}

reutrn 0;
}
```

```
int FAR PASCAL _export DeviceSetDriverParams(HDEVICE
hDevice, PDRIVERPARAMS pParams)
{
OutputDebugString("DeviceSetDriverParams \n");
if(!pParams)
    return -1;
if(!ValidHandle(hDevice))
    return -1;
if((hDevice->bFlags & FLAGS_OPEN)==0)
    return -1;

hDevice->params=*pParams;
return 0;
}

int FAR PASCAL _export DeviceGetDriverParams(HDEVICE
hDevice, PDRIVERPARAMS pParams)
{
OutputDebugString("DeviceGetDriverParams \n");
if(!pParams)
    return -1;
if(!ValidHandle(hDevice))
    return -1;
if((hDevice->bFlags & FLAGS_OPEN)==0)
    return -1;

*pParams -hDevice->params;
return 0;
}

int FAR PASCAL _export DeviceGetDriverCapabilities(HDEVICE
hDevice, PDRIVERPARAMS ppDriverCaps)
{
OutputDebugString("DeviceGetDriverParams \n");

if(!ppDriverCaps)
    return -1;
if(!ValidHandle(hDevice))
    return -1;
if((hDevice->bFlags & FLAGS_OPEN)==0)
    return -1;
```

```
*ppDriverCaps=&DriverCaps;
return 0;
}

BOOL ValidHandle(HDEVICE hDevice)
{
return (hDevice==&Device1);
}

BOOL Install_Ring0_Handler(DEVICECONTEXT *hDevice)
{
WORD sel;
BIMODAL_INT_STRUC *pBis;
WORD myds;

_asm
{
    xor  di,di
    mov  es,di
    mov  ax,0x1684
    mov  bx,VPICD_ID
    int  0x2F
    mov  WORD PTR [pfVPICDEntry],di
    mov  WORD PTR [pfVPICDEntry+2],es
    mov  ax,es
    or   ax,di
    jz   xit1

        mov  ax,VPICD_Get_Version
        call [pfVPICDEntry]
        cmp  ax,0x30A
    mov  WORD PTR [pfVPICDEntry],0
    mov  WORD PTR [pfVPICDEntry+2],0
xit1:
}
//Return FALSE if no Ring0 services
if(pfVPICDEntry==NULL)
    return (FALSE);

//fill in ring0 structure
pBis=&(hDevice->Ring0Info);
memset(pBis,0,sizeof(BIMODAL_INT_STRUC));
```

```
pBis→BIH_Irq_Number=hDevice→bIrq;
pBis→BIH_Enrey=OFFSETOF(Asm_Ring0_Isr);
pBis→BIH_Control_Proc=OFFSETOF(Asm_Ring0_Control);
pBis→BIH_User_Mode_CS=SELECTOROF(Asm_Ring0_Isr);
_asm mov  myda,ds
pBis→BIH_User_Mode_DS=myds;

pBis→BIH_DescrCount=2;
sel=SELECTOROF(hDevice→RxBuf.Start);
pBis→BIH_Descriptor[0].EBIS_User_Mode_Sel=sel;
sel=SELECTOROF(hDevice→TxBuf.Start);
pBis→BIH_Descriptor[1].EBIS_User_Mode_Sel=sel;

//Call VPICD to install Ring0 handler
-asm
{
mov   aax,VPICD_Install_Handler
mov   di,WORD PTR pBis
push  ds
pop   es
call  [pfVPICDEntry]
jnz   xit2 xor ax,ax
xit2:
}
return (TRUE);
}

VOID Remove_Ring0_Handler(DEVICECONTEXT *hDevice)
{
BIMODAL_INT_STRUC *pBis;
//Call VPICD remove Ring0 handler
pBis=&(hDevice→Ring0Info);
_asm
{
mov   ax,VPICD_Remove_Handler
mov   di,WORD PTR pBis
push  ds
pop   es
call  [pfVPICDEntry]
    }
}
/*End of File*/
```

列表 9.3 device.inc(设备信息)

```
BufInfo STRUC
Start        dd     0
BufSize       dw     0
IndexIn       dw     0
IndexOut      dw     0
BufInfo ENDS

DriverParams  STRUC
ReadBufSize   dw  0
DriverParams  ENDS

DeviceInfo  STRUC
DevNum          dw       0
IoBase          dw       0
Irq             db       0
Flags           db       0
Hwnd            dw       0
OldHandler      dd       0
RxBuf           db    SIZE BufInfo DUP(0)
TxBuf           db    SIZE BufInfo DUP(0)
Params          db    SIZE DriverParams DUP(0)
Numints         dw    0
; Do ont include Ring0 strucure here
DeviceInfo  ENDS
```

列表 9.4 ring0asm.inc(0 层处理器信息)

```
Ring0_Struc  STRUC
Ring0_IRQ_Number          dw    ?
Ring0_VM_ID               dw    ?
Ring0_Next                dd    ?
Ring0_Reserved1           dd    ?
Ring0_Reserved2           dd    ?
Ring0_Reserved3           dd    ?
Ring0_Reserved4           dd    ?
Ring0_Flags               dd    0
Ring0_Mode                dw    0
Ring0_Isr_Proc            dw    ?
Ring0_Control_Proc        dw    ?
                          dw    ?
Ring0_User_Mode_API       dd    ?
Ring0_Super_Mode_API      dd    ?
```

```
Ring0_User_Mode_CS        dw    ?
Ring0_User_Mode_DS        dw    ?
Ring0_Super_Mode_CS       dw    ?
Ring0_Super_Mode_DS       dw    ?
Ring0_Descriptor_Count    dw    ?
Ring0_Struc  ENDS

Ring0_Sel_Table           equ   word ptr (SIZE Ring0_Struc)

Ring0_Sel_Struc  STRUC
Ring0_User_Mode_Sel       dw    ?
                          dw    ?
Ring0_Super_Mode_Sel      dw    ?
Ring0_Sel_Struc  ENDS

Ring0_Sel1                equ   SIZE Ring0_Struc
Ring0_Sel2                equ   Ring0_Sel1+( SIZE Ring0_Sel_Struc)

.errnz   Ring0_Super_Mode_API - Ring0_SUser_Mode_API - 4
.errnz   Ring0_Super_Mode_CS - Ring0_SUser_Mode_CS - 4
.errnz   Ring0_Super_Mode_DS - Ring0_SUser_Mode_DS - 4
.errnz   Ring0_Super_Mode_Sel - Ring0_SUser_Mode_Sel - 4

VPICD_API_Get_Ver         equ   0000h
VPICD_Install_Handler     equ   0001h
VPICD_Remove_Handler      equ   0002h
VPICD_Call_At_Ring0       equ   0003h
VPICD_API_EOI             equ   0000h
VPICD_API_Mask            equ   0001h
VPICD_API_Unmask          equ   0002h
VPICD_API_Get_Mask        equ   0003h
VPICD_API_Get_IRR         equ   0004h
VPICD_API_Get_ISR         equ   0005h
VPICD_API_Call_Back       equ   0006h
```

列表 9.5　ring0asm.asm(0 层处理器编程)

```
include  DEVICE.INC
DATA SEGMENT WORD PUBLIC 'DATA'
DATA ENDS

INTERRUPT SEGMENT WORD PUBLIC 'CODE'
```

```
EXTRN _C_ring0_Usr:NEAR
EXTRN _C_CallBack_Event:NEAR

ASSUME cs:INTERRUPT

public _Asm_Ring0_Isr
_Asm_Ring0_Isr proc far
push  ds
push  si
push  ax
push  bx
push  cx
push  dx

; VPICD CALLS THIS ENTRY POINT WITH ES:DI POINTING TO OUR RING0 STRUCTURE.
; USE THIS TO GET ADDRESS OF DEVICEINFO STRUCTURE,
; SINCE WE LOCATED RING0 STRUCTURE AT THE END OF DEVICEINFO.
mov   si,es
mov   ds,si
lea   si,[di-SIZE DeviceInfo]

push  di
push  es
call  _C_Ring0_Isr

pop   es
pop   di
pop   dx
pop   cx
pop   bx
pop   ax
pop   si
pop   ds

_Asm_Ring0_Isr  endp

public  _Asm_Callback_Event
_Asm_Callback_Event proc far
mov   si,es
mov   ds,si
lea   si,[di-SIZE DeviceInfo]
```

```
call  _C_CallBack_Event
ret

_Asm_Callback_Event  endp

public _Asm_Ring0_Control
_Asm_Ring0_Control far
ret
_Asm_Ring0_Control endp

INTERRUPT  ENDS

END
; End  of  File
```

列表 9.6　ring0asm.h(0 层处理器编程)

```
void far cdecl Asm_Ring0_Isr(void);
void far cdecl Asm_Ring0_Control(void);
void far cdecl Asm_Ring0_Callback_Event(void);

#define  BIH_API_EOI        0x0000
#define  BIH_API_Call_Back   0x0006
/*End  of  File*/
```

列表 9.7　ring0isr.c(0 层处理器 C 函数编程)

```
#include <conio.h>
#include <windows.h>
#include "skeleton.h"
#include "uart.h"
#include "ring0asm.h"

DEVICECONTEXT *hDevice;
BYTE  intid, data, ier;
WORD  selRing0;
BYTE far *fpByte;
BIMODAL_INT_STRUC  *pBis;
VOIDFARPROC  pfApi;
VOIDFARPROC  pfCallback;

void near C_Ring0_Isr(void)
{
_asm mov hDevice, si;
```

```
pBis=&hDevice→Ring0Info;
pfApi=(VOIDFARPROC)pBis→BIH_Super_Mode_APT;

while(TRUE)
{
intid==_inp(hDevice→usIoBase+UART_REG_IIR);
if(intid==UART_IIR_NONE)
    break;
if(intid==UART_IIR_RXRDY)
{
    if((hDevice→RxBuf.In+1==hDevice→RxBuf.Out)
        ||(( hDevice→RxBuf.Out==0)&&
        (hDevice→RxBuf.In==hDevice→RxBuf.Size-1)))
    {
        pfCallback=(VOIDFARPROC)Asm_Callback_Event;
        _asm
                {
mov  ax,BIH_API_Call_Back
mov  bx,1
mov  cx,WORD FAR pfCallback+2
mov  dx,WORD FAR pfCallback
mov  si,1
push ds
pop  es
mov  di,pBis
call pfApi
                }
            }
            else
            {
data=inp(hDevice→usIoBase+UART_REG_RDR);
selRing0=pBis→BIH_Descriptor[0].EBIS_Super_Mode_Sel;
fpByte=MAKELP(selRing0,hDevice→RxBuf.In);
*fgByte=data;
hDevice→RxBuf.In++;
if(hDevice→RxBuf.In>=hDevice→RxBuf.Size)
    hDevice→RxBuf.In=0;
}
}
else if(intid==UART_IIR_THRE)
{
if(hDevice→TxBuf.In==hDevice→TxBuf.Out)
```

```
{
ier=_inp(hDevice→usIoBase+UART_REG_IER);
_outp(hDevice→usIoBase+UART_REG_IER,ier&(~UART_IER_THRE));
}
else
{
selRing0=pBis→BIH_Descriptor[1].EBIS_Super_Mode_Sel;
fpByte=MAKELP(selRing0,hDevice→TxBuf.In);
_outp(hDevice→usIoBase+UART_REG_THR,*fpByte);
hDevice→TxBuf.Out++;
if(hDevice→TxBuf.Out>=hDevice→TxBuf.Size)
   hDevice→TxBuf.Out=0;
}
}
}
pBis→BIH_Mode=0;
_asm
{
mov   ax,BIH_API_EOI
push  ds
pop   es
mov   di,pBis
call  pfApi
clc
}
}

void C_Callback_Event(void)
{
_asm  mov  hDevice,si
PostMessage(hDevice→hwnd,WM_USER,0,NULL);
}
/*End  of  File */
```

列表 9.8　ring0.def (0 层处理器定义)

```
LIBRARY       DRIVER
EXETYPE       WINDOWS
DATA          PRELOAD  FIXED  SINGLE
CODE          PRELOAD  FIXED  DISCARDABLE
```

列表 9.9　uart.h (0 层处理器定义)

```
#define  UART_REG_THR          0X00
```

```
#define  UART_REG_RDR        0X00
#define  UART_REG_IER        0X01
#define  UART_REG_IIR        0X02
#define  UART_REG_LCR        0X03
#define  UART_REG_MCR        0X04
#define  UART_REG_LSR        0X05
#define  UART_REG_BAUDLO     0X00
#define  UART_REG_BAUDHI     0X01

#define  UART_IIR_NONE       0X01
#define  UART_IIR_THRE       0X02
#define  UART_IIR_RXRDY      0X04
#define  UART_IER_THRE       0X02
#define  UART_IER_RXRDY      0X01
#define  UART_MCR_OUT2       0X08
#define  UART_MCR_LOOP       0X10
#define  UART_LSR_THRE       0X20
#define  UART_LCR_DLAB       0X80
#define  UART_LCR_8N1        0X03
#define  UART_ISR_RXRDY      0X01
#define  BAUD_1200           0X60

//END  OF  FILE
```

列表 9.10　ring0.mak (0 层处理器标志)

```
all: driver.dll

#DRIVER  DLL

#non-interrupt code(.C)
skeleton.obj:skeleton.c skeleton.h
  cl -c -W3 -ASw -Gsw2 -Zi -Oi -Fc $*.c

#interrupt code(.C)
ring0isr.obj: ring0isr.c skeleton.h ring0asm.h
  cl -c -W3 -ASw -Gsw2 -Zi -Oi -Fc $*.c

#interrupt code(.ASM)
ring0asm.obj: ring0asm.asm device.inc
  ml -c -W2 -Cp $*.asm

driver.dll: driver.def skeleton.obj ring0isr.obj ring0asm.obj
```

```
  link  skeleton+ ring0isr+ ring0asm.obj, driver.dll, driver.map/CO /MAP,
\
sdllcew libw/nod/noe, driver.def
  mapsym driver
  implib  driver.lib  driver.dll driver .def
  rc driver.dll
  copy  driver.dll  \win31
```

第 10 章　双模态驱动程序

前面已经讨论过在运行 Windows 时，微处理器经常要切换状态。在 Windows 的增强模式下，微处理器在第 0 层保护模式（监控管理程序）、第 3 层保护模式（Windows 应用程序)以及 V86 模式间进行切换(DOS 和 BIOS 服务及 DOS 应用程序)。在标准模式 Windows 下工作就简单一些，只在第 3 层保护模式（Windows 应用程序）和实模式两种模式间建立切换（DOS 和 BIOS 服务及 DOS 应用程序）。

这些经常性的切换导致了大量的中断等待时间。上一章讲述了在增强模式下将这个等待时间减到最小的方法，本章将讨论在标准模式下将这个等待时间减到最小的方法。

如果微处理器在保护模式时发生中断，Windows 的驱动程序可以在不需要切换的前提下服务中断，因为 Windows 驱动程序运行在保护模式中。Windows 驱动程序也可以不用采取特殊的动作来捕获实模式中发生的中断，因为 Windows 自动将接收到的实模式中断反射到保护模式处理器。

然而，这个反射需要在实模式和保护模式之间切换。这对于运行在标准模式的 Windows 可能是个问题，因为这个模式切换的代价是非常大的。令人吃惊的是，尽管有一个从实模式切换到保护模式的指令，但是没有从保护模式切换到实模式的指令。Windows 必须实际地复位，才能回到时模式。这个保护模式转换到实模式的时间可能需要几百个微秒到一毫秒，而在这期间不会对中断服务。

如果某些设备不能忍受这样的中断延时，而驱动程序又要运行在标准模式中，那么就需要双模态处理器。双模态处理器可以运行在任何一个中断发生的模式中，而不用在模式切换上耗费时间。

10.1　新的双模态驱动程序综述

本章的例子驱动程序是在第 4 章基础的中断驱动程序上修改而来的。驱动程序 DLL 的入口点保持一样，所以它可以用同一个 TESTDRIV 应用程序测试。唯一不同的是当在 Windows 标准模式下运行时，驱动程序在保护模式和实模式中分别处理一些中断，这依赖于在中断发生时微处理器的状态。

有 3 个关键字使中断处理器运行在实模式中，将有关的驱动程序组件放在低端内存中，以便它们可以在实模式中寻址；在实模式中从中断处理器访问数据；从实模式调用 WindowsAPI 函数。

这 3 个关键字需要通过以下一些步骤来将基础的中断驱动程序转换到双模态的中断驱动：

1．在中断时将使用的代码和数据放到 1MB 以下。

2. 修改中断处理器来加载 DS。
3. 安装实模式处理器。
4. 确定在实模式中映射驱动程序数据段的段值。
5. 修改中断处理器来获取微处理器的模式。
6. 修改中断处理器来在实模式中通过回调来调用 WindowsAPI。

10.1.1 新的双模态驱动程序代码

将基础的中断驱动程序转换到双模态的中断驱动步骤，只需通过修改 DeviceOpen（）函数，DeviceClose（）函数和中断处理器本身就能实现。

1. 将程序放在 1MB 空间以下

前面的基础中断驱动程序要求中断时使用的代码和数据是固定、不可丢弃和页面锁定的。双模态驱动程序也有这些要求。但是为了在实模式中运行，双模态驱动程序还有一个附加的条件：它的中断代码和数据不能存在实模式中可寻址的地方，就是说它们必须放在 1MB 界限以下。WindowsAPI 函数 GlobalDosAlloc（）确保分配一块位于 1MB（线性地址）以下的区域。在函数 DeviceOpen（）中动态分配的缓冲区满足这个条件，因为它们是通过 GlobalDosAlloc（）分配的。

基础驱动程序对它的代码和数据段的要求可以通过在模块定义文件中放置关键字来实现。但是没有一个关键字能迫使 Windows 将一个段放在 1MB 以下。双模态驱动程序可以采取的最好办法是在代码和数据段都使用 FIXED，并且希望 Windows 将它们放到 1MB 以下。当它运行的时候，驱动程序检查是否需要将这些段放到 1MB 以下，如果是，驱动程序通过分别安装实模式处理器和保护模式处理器来实现双模态；如果不是，驱动程序则不需要双模态，而只安装一个处理器。

在大多数情况下这个办法工作得都非常好， Windows 内存管理器分配 FIXED 内存。Windows 内存管理器使用的策略是从堆的底部分配 FIXED 内存，从堆的上部分配 DISCARDABLE 内存，在中间分配 MOVEABLE 内存。由于堆在 DOS 程序加载区域开始，低于 640KB，所以大部分 FIXED 内存分配在 1MB 以下。

2. 加载 DS

一些中断处理器必须和处理器的数据段一起调入 DS。为了保证双模态，处理器必须在保护模式中将一个选择器调入 DS，而在实模式中将一个段调入 DS。也就是说，处理器必须在它运行时知道微处理器的模式，以便可以决定是调入一个选择器还是一个段。

双模态驱动程序通过使用两个不同的入口点进入处理器来实现它。每个入口点调入相应的 DS，并调用一个通用的处理器 C_CommonHandler（）来实际服务中断。在保护模式中调用入口点将选择器调入 DS，而在实模式中调用入口点将段调入 DS。

3. 安装实模式处理器

双模态驱动程序使用前面讲过的 DOS 设置向量来安装保护模式（与基础驱动程序使用的中断服务相同）。安装使用这个机制的处理器将在保护模式中一直被调用。这个保护模式的处理器经常被调用来处理实模式中发生的中断，除非 Windows 关心模式切换，并

且将必要的选择器加载到段寄存器。即是说，驱动程序需要知道中断发生时的实际模式，避免了因为模式切换带来的巨大代价，这也是双模态处理器的优点。

驱动程序还可以通过 DPMI 设置实模式向量服务来安装实模式处理器。如果已经安装了这样的处理器，Windows 会调用安装处理器而不是切换微处理器的模式来反射实模式的中断。

4．确定实模式段

在实模式处理器调入到 DS 段中时，段在运行时通过将保护模式的选择器转换成相对应的实模式的段来确定。可以通过调用 Windows API 函数 GetSelectorBase（）很容易地实现。

5．修改中断处理器

（1）DEVICECONTEXT 的改变

DEVICECONTEXT 和 BUFINFO 结构在双模态驱动程序中都有所扩充。DEVICE CONTEXT 增加了一个新字段 pfOldRealHandler，用以存储原始实模式处理器的地址。另外，基础驱动程序中的 pfOldHandler 在双模态驱动程序中被改为 pfOldProtHandler。

BUFINFO 结构也被修改了。Start 字段被改名为 ProtModeStart，并增加了一个新的字段 RealModeStart。ProtModeStart 存储向量，而 RealModeStart 存储段。因为层缓冲区的输入和输出是索引，而不是指针，所以它们不需要为了在实模式运行而修改。

（2）DeviceOpen（）的改变

双模态驱动程序中的 DeviceOpen（）函数分配传送和接收缓冲区，安装中断处理器，并且初始化串口（与基础中断驱动程序一样）。但是在细节上有所不同。双模态驱动程序使用 GlobalDosAlloc（）代替 GlobalAlloc（）分配缓冲区，以便缓冲区可以在实模式下寻址。GlobalDosAlloc（）返回一个实模式的段和一个屏蔽模式的向量，并且 DeviceOpen（）将它们分别存储在 RealModeStart 和 ProtModeStart 中。

DeviceOpen（）首先通过 DOS 的设置向量来安装保护模式入口点。接着调用 GetWind Flags（）来查明 Windows 是否在标准模式中。如果不是，实模式处理器就不必要了，双模态驱动程序变为与第 4 章的基础中断驱动程序一样。

如果 Windows 在标准模式中，DeviceOpen（）调用 DPMI 的设置实模式向量来安装实模式入口点。DPMI 设置实模式中断向量需要处理器的实模式地址。DeviceOpen（）使用 GetSelectorBase（）将处理器的保护模式地址转换为实模式地址，其中选择器是 CS。在实模式中，物理地址是通过将段值左移 4 位得到的。所以要从物理地址得到实模式段值，只需要将其右移 4 位即可。

接着，DeviceOpen（）决定它的数据段的实模式地址，并存储起来以便以后中断处理器使用。中断处理器以及驱动程序的其他部分共享相同的地址段，所以我们需要在 DS 中的实模式地址。然后再一次使用 GetSlectorBase（）将保护模式选择器（这次是 DS）转换为相应的实模式的值。

在计算实模式的值之后，DeviceOpen（）将这个值传到函数 SaveThisInHandler（）。在汇编语言模块中分配的 SaveThisInHandler（）实际存储这个值。

这个函数调用的原因是用来存储实模式段值的变量 RealModeDS 实际在处理器模块中

分配。在数据段中，将 RealModeDS 声明为普通的变量对中断处理器并没有什么好处，因为处理器只有在加载 DS 后才能访问数据段。所以 RealModeDS 直接在中断处理器的代码段中声明。由于中断处理器是一个汇编语言模块，所以很容易实现。

SaveThisInHandler（）按位处理，因为 Windows 不允许直接写代码段。为了实现它，SaveThisInHandler（）使用没有正式说明的 Windows API 函数 AllocCSToDSAlias（）。AllocCSToDSAlias（）只有一个参数——代码选择器，并返回一个严格映射到相同区域的数据选择器。

SaveThisInHandler（）将这个别名选择器加载到 ES 中，并将实模式值的参数存储到 RealModeDS 中。最后别名选择器通过 FreeSelector（）释放。在 DeviceOpen（）中初始化串口的代码，这一点与基础中断驱动程序相同。

（3）DeviceClose（）的改变

与前面相比，DeviceClose（）执行与 DeviceOpen（）相反的活动：禁止设备中断，卸载中断处理器，并释放缓冲区。设备控制代码不需要改变。双模态的 DeviceClose（）同时存储原始的保护模式处理器和原始的实模式处理器。以前 GlobalDosAlloc（）分配的缓冲区通过调用 GlobalFree（）来释放掉。

（4）新的中断处理器入口点

第 4 章所讲的基础中断驱动程序的中断处理器，完全是用 C 语言中的关键字 interrupt 来写的。这种方法在双模驱动程序中不能工作。本章的驱动程序包括一个独立的汇编语言模块 BIMODAL.ASM。两个中断处理器的入口点（ProtHandler 和 RealHandler）就在这个模块中。下面讲一下为什么使用汇编语言。

在 C 语言函数中，关键字 interrupt 执行下面的指令来对 DS 进行加载：

```
mov   ax,DATA
mov   ds,ax
```

因为当程序被调入到内存后才能知道实际的数据段，所以关键字 DATA 在链接时将产生一个临时的填充值。加载器将程序的代码段和数据段调入到内存后，立即将实际的段值放入到这个填充值中。当驱动程序的 DLL 被加载时，这个段值就是保护模式选择器。这样如果中断处理器在保护模式中被调用，那么前面提到的指令就能正确地加载 DS。

但是如果中断处理器在实模式中被调用，就不能保证 DS 加载正确的值，双模态驱动程序处理器不能用 C 语言使用关键字 interrupt 来实现。所以将实模式和保护模式的入口点都放在汇编语言模块 BIMODAL.ASM 中。

实模式入口点的代码 RealHandler（）非常短。首先，它将所有的寄存器存储起来，并从变量 RealModeDS 加载 DS（由 DeviceClose（）初始化）。接着它调用 C_CommonHandler（）实际服务中断，并传递值为 MODE_REAL 的参数。最后 RealHandler（）有 IRET 返回。唯一不平常的技术就是段不访问 RealModeDS，因为 RealModeDS 已在代码段中声明。

```
RealHandler  PROC  NEAR
pusha
push    ds
```

```
mov     ax,cs:[RealModeDS]
mov     ds,ax
push    MODE_REAL
call    C_CommonHandler
pop     ds
popa
iret
```

保护模式入口点 ProtHandler 的代码与它类似。ProtHander 使用关键字 DATA 加载 DS，由 MODE_PROT 调用 C_CommonHandler，由 IRET 返回。

```
ProtHandler  PROC  NEAR
pusha
push    ds
mov     ax,DATA
mov     ds,ax
push    MODE_PROT
call    C_CommonHandler
pop     ds
popa
iret
```

（5）改变为 C 语言的中断处理器函数 C_CommonHandler（）从技术上讲不是真的中断处理器，因为它不是当作处理器安装的，而且不返回 IRET，所以在基础中断驱动程序中的关键字 interrupt 在 C_CommonHandler（）中不能使用。C_CommonHandler（）在某些方面确实是中断处理器，因为它是服务于设备中断的函数。服务设备的大部分代码在状态上是独立的。在基础中断驱动程序的 DeviceIsr 中稍作改动就成了双模态驱动程序中的 C_CommonHandler（）了。

改动之一包括在两个层缓冲区中访问数据，它们不在驱动程序的数据段中。C_CommonHandler（）在保护模式过程中使用一个选择器来访问缓冲区，但在实模式中断过程中是实模式段。如果模式参数是 MODE_PROT，BUFINFO 的 StartProt 字段被用来构成指针；如果模式参数是 MODE_REAL，则 BUFINFO 的 StartReal 字段被用来构成指针。

另一个变化在调用 PostMessage（）中。处理器在接收层缓冲区溢出时向应用程序的消息队列发出一个用户定义的消息。若传递给 C_CommonHandler（）的模式参数是 MODE_PROT，处理器就在保护模式中执行（就像 Windows 的 API 函数 PostMessage（）一样），并且处理器直接调用 PostMessage（）。如果模式参数是 MODE_REAL，处理器就不能直接调用 PostMessage（），因为微处理器在实模式中，而 PostMessage（）不能在实模式中工作。处理器必须安排一个在保护模式中的回调，并从那里调用 PostMessage（）。这并不很复杂，因为 DPMI 服务 Allocate Real Mode Call-Back Address 恰好提供了需要。

10.1.2　回调细节

在表 10-1 中列出了 DPMI 的 Allocate Real Mode Call-Back Address 服务参数。要分配 DPMI 的回调，必须给 DPMI 两方面的信息。第一，DS：SI 保护模式中函数的地址。在 C_CommonHandler（）中，是 BIMODAL.ASM 中的 ProtModeCallBack。另外，在 ES：DI 中是实模式调用结构。图 10-1 所示的实模式调用结构中每个寄存器都有一个字段。这个结构将环境的信息提供给你的保护模式函数。C_CommonHandler（）使用实模式调用结构的 DS 和 IS 字段来存储指向它的 DEVICECONTEXT 的指针。注意存于 DS 字段中的是一个保护模式选择器，因为回调发生在保护模式中。

偏移量	寄存器	偏移量	寄存器
00h	EDI	22h	ES
04h	ESI	24h	DS
08h	EBP	26h	FS
0Ch	系统保留	28h	GS
10h	EBX	2Ah	IP
14h	EDX	2Ch	CS
18h	ECX	2Eh	SP
1Ch	EAX	30H	SS

图 10-1　DPMI 分配回调所需的实模式调用结构

表 10-1　分配实模式回调地址

调用参数	
AX	0303H
DS:(E)SI	选择器：保护模式过程的偏移量
ES:(E)DI	选择器：实模式寄存器数据结构中 32 字节的偏移量
结果	
Carry	如果成功，就清 0；否则，置 1
CX:DX	段：回调的偏移量

DPMI 的 Allocate Real Mode Call-Back Address 在 CX:DX 中返回一个函数指针。这个函数在调用时将通过 DS：SI 得到 DPMI 地址的保护模式的异步回调。但是由于回调是异步的，所以 C_CommonHandler（）调用 DPMI 的地址，接着完成中断的处理并返回。

在这以后，BIMODAL.ASM 中的 ProtModeCallback（）被执行，但这时微处理器在保护模式中。当进入回调时，ES：DI 指向一个以前在 DPMI 注册的实模式回调结构。ProtModeCallback（）从回调结构的 DS 和 SI 字段中提取 DEVICECONTEXT 的指针，从 DEVICECONTEXT 检索 Windows 的控制，并调用 PostMessage（）。

当 ProtModeCallback（）返回和 DPMI 从保护模式切回到实模式时，DPMI 从实模式调用结构中调入实模式寄存器的值。允许通过修改实模式调用结构来使保护模式代码与实模式代码进行通信。不过，同时 DPMI 从调用结构调入 CS 和 IP，并且使用保护模式代码来正确固定 CS 和 IP。

这也不像听起来的那样难。除了在 ES:DI 中传递的实模式回调结构环境外，DPMI 还提供指向实模式堆栈的指针。并且实模式堆栈包含实模式的返回地址。所以，ProtModeCallback（）采取的下一个动作就是将返回的地址从实模式堆栈拷贝到回调结构的 CS:IP 字段中。

10.2 总 结

如果性能敏感的驱动程序被标准模式用户所需要，那么双模态驱动程序是最好的选择。由于双模态处理器必须处理几个涉及到保护模式和实模式之间区别的细节（如本章所示），这些问题不需要将驱动程序复杂化。

列表 10.1 bimodal.h(双模式驱动程序)

```
#include "rmcs.h"
#define FALGS_ON_SLAVE_PIC    0x01
#define FALGS_OPEN            0x02
#define FALGS_RXQOVER         0x04
#define FALGS_BIMODAL         0x08

#define MSATER_PIC_CTRL       0x20
#define MSATER_PIC_MASK       0x21
#define SLAVE_PIC_CTRL        0xA0
#define SLAVE_PIC_MASK        0xA1
#define EOI 0x20

#define SET(value,mask)      value|=mask
#define CLR(value,mask)      value&=(~mask)

typedef struct
{
WORD usReadBufSize;
}DRIVERCAPS, FAR *PDRIVERCAPS;

typedef  PDRIVERCAPS FAR * PDRIVERCAPS

typedef void(FAR interrupt *VOIDINTPROC)();
typedef void(FAR *VOIDINTPROC)()
```

```
typedef struct
{
char  far *ProtModeStart;
   char  far *RealModeStart;
WORD  Size;
WORD  In;
WORD  Out;
}BUFINFO;

typedef struct
{
WORD                    usDevNuMBer;
WORD                    usIoBase;
BYTE                    bIrq;
BYTE                    bFlags;
HWND                    hwnd;
VOIDINTPROC             pfOldProtHandler;
VOIDINTPROC             pfOldRealHandler;
BUFINFO                 RxBuf;
BUFINFO                 TxBuf;
WORD                    numIntsReal;
WORD                    numIntsProt;
DRIVERPARAMS            params;
RMCS                    rmcs;
FARPROC                 pfCallback;
}DEVICECONTEXT, FAR *HDEVICE;

HDEVICE FAR PASCAL DeviceOpen(HWND hwnd);
int FAR PASCAL DeviceClose(HDEVICE);
int FAR PASCAL DeviceGetWriteStatus(HDEVICE, LPWORD pusStatus);
int FAR PASCAL DeviceGetReadStatus(HDEVICE, LPWORD pusStatus);
int FAR PASCAL DeviceWrite(HDEVICE, LPBYTE lpData, LPWORD pcBytes);
int FAR PASCAL DeviceRead(HDEVICE, LPBYTE lpData, LPWORD pcBytes);
int FAR PASCAL DeviceSetDriverParams(HDEVICE, PDRIVERPARAMS pParams);
int FAR PASCAL DeviceGetDriverParams(HDEVICE, PDRIVERPARAMS pParams);
int FAR PASCAL DeviceSetDriverCapabilities(HDEVICE, PPDRIVERCAPS
ppDriverCaps);

extern DEVICECONTEXT Device1;
/*End of file.*/
```

列表 10.2　uart.h

```
#define  UART_REG_THR         0X00
#define  UART_REG_RDR         0X00
#define  UART_REG_IER         0X01
#define  UART_REG_IIR         0X02
#define  UART_REG_LCR         0X03
#define  UART_REG_MCR         0X04
#define  UART_REG_LSR         0X05
#define  UART_REG_BAUDLO      0X00
#define  UART_REG_BAUDHI      0X01

#define  UART_IIR_NONE        0X01
#define  UART_IIR_THRE        0X02
#define  UART_IIR_RXRDY       0X04
#define  UART_IER_THRE        0X02
#define  UART_IER_RXRDY       0X01
#define  UART_MCR_OUT2        0X08
#define  UART_MCR_LOOP        0X10
#define  UART_LSR_THRE        0X20
#define  UART_LCR_DLAB        0X80
#define  UART_LCR_8N1         0X03
#define  UART_ISR_RXRDY       0X01
#define  BAUD_1200            0X60
#define  BAUD_110             0X0417L

//END  OF  FILE
```

列表 10.3　bimodal.h

```
typedef struct
{
DWORD edi;
DWORD esi;
DWORD ebp;
DWORD reserved;
DWORD ebx
DWORD edx;
DWORD ecx;
DWORD eax;
WORD  falgs;
WORD  es;
WORD  ds;
WORD  fs;
```

```
WORD  gs;
WORD  Ip;
WORD  cs;
WORD  sp;
WORD  ss;
}RMCS;
/* End  of  File */
```

列表 10.4　rtmcs.c

```
#define _WINDLL
#define STRICT

#include <dos.h>
#include <conio.h>
#include <windows.h>
#include <windowsx.h>
#include "skeleton.h"
#include "uart.h"
#include "bimodal.h"

#define DOS_GET_INT_VECTOR            0x35
#define DOS_SET_INT_VECTOR            0x25

#define DPMI_GET_SEB_BASE             0x0006
#define DPMI_GET_REAL-INT VECTOR      0x0200
#define DPMI_SET_REAL-INT VECTOR      0x0201
#define DPMI_ALLOC_RM_CALLBACK        0x0303
#define DPMI_FREE_RM_CALLBACK         0x0304
#define DPMI_GET_VERSION              0x0400

#define REAL_MODE_CEILING             0x00100000L

DEVICECONTEXT  Device1(0,0x3F8,4,0,NULL);
DRIVERPARAMS   DefaultParams(1024);
DRIVERPARAMS   DriverCaps(0x0101);

UINT WINAPI AllocCStoDSAlloc(UINT);

BOOL ValidHandle(HDEVICE hDevice);
VOIDINTPROC DosGetIntVector(BYTE Irq);
void DosSetIntVector(BYTE Irq, VOIDINTPROC pHandler);
VOIDINTPROC DPMI_GetRMVector(BYTE Irq);
```

```
void DPMI_SetRMVector(BYTE Irq, VOIDINTPROC pHandler);
void DPMI_AllocateRMCallback(FARPROC pfCallThisFunc,RMCS far *pRmcs,
                FARPROC far *ppfCallbackAddr);
void DPMI_FreeRMCallback(FARPROC pfCallbackAddr);
WORD SelectorToSegment(WORD Sel);

HDEVICE FAR PASCAL DeviceOpen(HWND hwnd)
{
HDEVICE  hDevice;
BYTE     bVector,mask;
DWORD    dw,winflags;
BYTE     baseIntMaster, baseIntSlave;
WORD     ds,cs,rmseg;

OutputDebugString("DeviceOpen\n");
HDevice=&Device1;
if(hDevice→bFlags & FLAGS_OPEN)
{
   OutputDebugString("Eoorr already Open\n");
      return (HDEVICE)-1;
}
hDevice→params=DefaultParams;

  hDevice→RxBuf.Size= hDevice→params.usReadBufSize;
dw=GlobalDosAlloc(hDevice→RxBuf.Size);
if!(dw==NULL)
{
   OutputDebugString("ERROR GlobalAlloc Rx\n");
   Return (HDEVICE)-1;
}
hDevice→RxBuf.In= hDevice→RxBuf.Out=0;

  hDevice→TxBuf.Size= hDevice→params.usReadBufSize;
dw=GlobalDosAlloc(hDevice→TxBuf.Size);
if(!dw)
{
   OutputDebugString("ERROR GlobalAlloc Tx\n");
   Return (HDEVICE)-1;
}
hDevice→TxBuf.In= hDevice→TxBuf.Out=0;
hDevice→hwnd=hwnd;
```

```
    if(hDevice→bIrq<8)
    {
    mask=_inp(MASTER_PIC_MASK);
    SET(mask,1<<hDevice→bIrq);
    _outp(MASTER_PIC_MASK,mask);
    }
else
{
    SET(hDevice→bFlags,FLAGS_ON_SLAVE_PIC);
    mask=_inp(SLAVE_PIC_MASK);
    SET(mask,1<<hDevice→bIrq-8);
    _outp(SLAVE_PIC_MASK,mask);
    }
if(hDevice→bIrq<8)
bVector= hDevice→bIrq+0x08;
else
    bVector= hDevice→bIrq-8+0x70;

hDevice→pfOldProtHandler=DosGetIntVector(bVector);
DosSetIntVector(bVector,(VOIDINTPROC)ProtHandler);

_asm mov _cs,cs;
_asm mov _ds,ds;
winflags=GetWinFlags();
if(winflags & WF_STANDARD)&&(GetSelectorBase(_cs)<REAL_MODE_CEILING)
                && GetSelectorBase(_ds)<REAL_MODE_CEILING))
{
      SET(hDevice→bFlags,FLAGS_BIMODAL);
    _asm
    {
    mov  ax,DPMI_GET_VERSION
    int  31h
    mov  baseIntMaster,dh
    mov  baseIntSlave,dl
}
    if(hDevice→bIrq<8)
    bVector= hDevice→bIrq+baseIntMaster;
    else
        bVector= hDevice→bIrq+ baseIntSlave;
hDevice→pfOldProtHandler=DPMI_GetRMVector(bVector);
rmseg=SelectorToSegment(  SELECTOROF(RealHandler));
PMI_SetRMVector(bVector,MAKELP(rmseg,OFFSETOF(RealHandler)));
```

```
_asm mov _ds,ds
SaveThisHandler(SelectorToSegment(_ds));
DPMI_AllocateRMCallback(ProtModeCallback,&hDevice→rmcs,
             &hDevice→pfCallback);
}

//Configure UART
_outp(hDevice→usIoBase+UART_REG_IER,0);
_outp(hDevice→usIoBase+UART_REG_LCR,UART_LCR_DLAB);
_outp(hDevice→usIoBase+UART_REG_BAUDLO,BAUD_1200);
_outp(hDevice→usIoBase+UART_REG_BAUDNI,0);
_outp(hDevice→usIoBase+UART_REG_LCR,UART_LCR_8N1);
_outp(hDevice→usIoBase+UART_REG_IER,UART_IER_RXRDY);
_outp(hDevice→usIoBase+UART_REG_MCR,UART_MCR_OUT2);

//Unmask interrupt at pic
if(hDevice→bIrq<8)
{
   mask=_inp(MASTER_PIC_MASK);
   CLR(mask,1<<hDevice→bIrq);
   _outp(MASTER_PIC_MASK,mask);
}
else
{
   mask=_inp(SLAVE_PIC_MASK);
   CLR(mask,1<<hDevice→bIrq-8);
   _outp(SLAVE_PIC_MASK,mask);
}

SET(hDevice→bFlags,FLAGSOPEN);
return hDevice;
}

int FAR PASCAL _export DeviceClose(HDEVICE hDevice)
{
BYTE mask;
OutputDebugString("DeviceClose\n");
if(!ValidHandle(hDevice))
   return -1;
if((hDevice→bFlags & FLAGS_OPEN)==0)
   return FALSE;
CLR(hDevice→bFlags,FLAGS_OPEN);
```

```
//disable UART interrupt
_outp(hDevice->usIoBase+UART_REG_IER,0);
_outp(hDevice->usIoBase+UART_REG_MCR,0);

if(hDevice->bIrq<8)
{
    mask=_inp(MASTER_PIC_MASK);
    SET(mask,1<<hDevice->bIrq);
    _outp(MASTER_PIC_MASK,mask);
}
else
{
    SET(hDevice->bFlags,FLAGS_ON_SLAVE_PIC);
    mask=_inp(SLAVE_PIC_MASK);
    SET(mask,1<<hDevice->bIrq);
    _outp(SLAVE_PIC_MASK,mask);
}

if(hDevice->bIrq<8)
bVector= hDevice->bIrq+0x08;
else
    bVector= hDevice->bIrq-8+0x70;

DosSetIntVector(bVector,hDevice->pfOldProtHandler);
if(hDevice->bFlags & FLAGS_BIMODAL)
{
    DPMI_SetRMVector(bVector, hDevice->pfOldProtHandler);
    DPMI_FreeRMCallback(hDevice->pfCallback);
}
GlobalFreePtr(hDevice->RxBuf.ProtModeStart);
GlobalFreePtr(hDevice->TxBuf.ProtModeStart);

return 0;
}

int FAR PASCAL DeviceGetWriteStatus(HDEVICE hDevice,LPWORD pusStatus)
{
OutputDebugString("DeviceGetWriteStatus\n");
if(!ValidHandle(hDevice))
    return -1;
if((hDevice->bFlags & FLAGS_OPEN)==0)
```

```
    return -1;

if(_inp(hDevice→usIoBase+UART_REG_LSR) & UART_LSR_THRE)
    *pusStatus=1;
else
    *pusStatus=0;

return 0;
}

int FAR PASCAL DeviceGetReadStatus(HDEVICE hDevice,LPWORD pusStatus)
{
OutputDebugString("DeviceGetReadStatus\n");
if(!ValidHandle(hDevice))
    return -1;
if((hDevice→bFlags & FLAGS_OPEN)==0)
    return -1;

if(hDevice→RxBuf.In!=hDevice→RxBuf.Out)
    *pusStatus=1;
else
    *pusStatus=0;

return 0;
}

int FAR PASCAL DeviceWrite(HDEVICE hDevice,LPBYTE lpData,LPWORD pcBytes)
{
WORD i;
BYTE ier;

OutputDebugString("DeviceWrite\n");
if(!lpData)
{
    OutputDebugString("ERROR\n");
    return -1;
}
if(!ValidHandle(hDevice))
{
    OutputDebugString("ERROR\n");
    return -1;
}
```

```
if((hDevice→bFlags & FLAGS_OPEN)==0)
{
    OutputDebugString("ERROR\n");
    return -1;
    }

for(i=0; i<*pcBytes; i++)
{
    hDevice→TxBuf.ProtModeStart[hDevice→TxBuf.In++]=lpData[i];
    if(hDevice→TxBuf.In>=hDevice→TxBuf.Size)
        hDevice→TxBuf.In=0;
}

if(UART_LSR_THRE & _inp(hDevice→usIOBase+UART_REG_LSR))
{
    ier=_inp(hDevice→usIoBase+UART_REG_LER);
    if((UART_IER_THRE & ier)==0)
    {
        _outp(hDevice→usIoBase+UART_REG_LER,ier| UART_IER_THRE);
    }
}

return 0;
}

int FAR PASCAL DeviceRead(HDEVICE hDevice,LPBYTE lpData,LPWORD pcBytes)
{
WORD i,cBytes;

OutputDebugString("DeviceRead\n");
if(!lpData)
{
    OutputDebugString("ERROR\n");
    return -1;
}
if(!ValidHandle(hDevice))
{
    OutputDebugString("ERROR\n");
    return -1;
}
if((hDevice→bFlags & FLAGS_OPEN)==0)
{
```

```
    OutputDebugString("ERROR\n");
    return -1;
    }

cBytes=*pcBytes;
for(i=0; i<cBytes; i++)
{
    if(hDevice→RxBuf.In>=hDevice→RxBuf.Out)
        break;
    lpData[i]=hDevice→RxBuf.ProtModeStart[hDevice→RxBuf.Out++];
    if(hDevice→RxBuf.Out>=hDevice→RxBuf.Size)
        hDevice→RxBuf.Out=0;
    *pcBytes--;
}
reutrn 0;
}

int FAR PASCAL _export DeviceSetDriverParams(HDEVICE
hDevice, PDRIVERPARAMS pParams)
{
OutputDebugString("DeviceSetDriverParams \n");
if(!pParams)
    return -1;
if(!ValidHandle(hDevice))
    return -1;
if((hDevice→bFlags & FLAGS_OPEN)==0)
    return -1;

hDevice→params=*pParams;
return 0;
}

int FAR PASCAL _export DeviceGetDriverParams(HDEVICE
hDevice, PDRIVERPARAMS pParams)
{
OutputDebugString("DeviceGetDriverParams \n");
if(!pParams)
    return -1;
if(!ValidHandle(hDevice))
    return -1;
if((hDevice→bFlags & FLAGS_OPEN)==0)
    return -1;
```

```
*pParams =hDevice→params;
return 0;
}

int FAR PASCAL _export DeviceGetDriverCapabilities(HDEVICE
hDevice,PDRIVERPARAMS ppDriverCaps)
{
OutputDebugString("DeviceGetDriverParams \n");

if(!ppDriverCaps)
    return -1;
if(!ValidHandle(hDevice))
    return -1;
if((hDevice→bFlags & FLAGS_OPEN)==0)
    return -1;

*ppDriverCaps=&DriverCaps;
return 0;
}

BOOL ValidHandle(HDEVICE hDevice)
{
return (hDevice==&Device1);
}

VOIDINTPROC DosGetIntVector(BYTE vector)
{
WORD selHandler,offHandler;
_asm
{
mov  al,vector
mov  ah,DOS_GET_INT_VECTOR
push es
int  21h
mov  offHandler,bx
mov  selHandler,es
pop  es
}
return(MAKELP(selHandler,offHandler));
}
```

```
void DosSetIntVector(BYTE vector,VOIDINTPROC pHandler)
{
WORD selHandler,offHandler;
selHandler=SELECTOROF(pHandler);
offHandler=OFFSETOF(pHandler);
_asm
{
mov  al,vector
mov  ah,DOS_GET_INT_VECTOR
mov  dx,offHandler
mov  bx,selHandler
push ds
mov  ds,bx
int  21h
pop  ds
}
}

WORD SelectorToSegment(OWRD Sel)
{
LONG base;
base=GetSelectorBase(Sel);
return((WORD)(base>>4));
}

VOIDINTPROC DPMI_GetRMSelector(BYTE vector)
{
WORD selHandler,offHandler;
_asm
{
mov  ax,DPMI_GET_REAL_INT_VECTOR
mov  bl,vector
int  31h
mov  offHandler,cx
mov  selHandler,dx
}
return(MAKELP(selHandler,offHandler));
}

void DPMI_SetRMSelector(BYTE vector,VOIDINTPROC pHandler)
{
WORD selHandler,offHandler;
```

```
SelHandler=SELECTOROF(pHandler);
OffHandler-OFFSETOF(pHandler);

_asm
{
mov  ax,DPMI_SET_REAL_INT_VECTOR
mov  bl,vector
mov  cx,offHandler
mov  dx,selHandler
int  31h
}
}

void DPMI_AllocateRMCallback(FARPROC pfCallThisFunc,RMCS far *pRmcs,
                FARPROC far *ppfCallbackAddr)
{
WORD selCallThisFunc,offCallThisFunc;
WORD selRmcs,offRmcs;
WORD selCallbackAddr,offCallbackAddr;
selCallThisFunc =SELECTOROF(pfCallThisFunc);
offCallThisFunc =OFFSETOF(pfCallThisFunc);
selRmcs =SELECTOROF(pRmcs);
offRmcs =OFFSETOF(pRmcs);
_asm
{
mov  ax,DPMI_ALLOC_RM_CALLBACK
mov  si, selCallThisFunc
push ds
mov  ds,si
mov  si, offCallThisFunc
mov  di,selRmcs
push es
mov  es,di
mov  di, offRmcs
int  31h
pop  es
pop  ds
mov  offCallbackAddr,cx
mov  selCallbackAddr,dx
}
*ppfCallbackAddr=MAKELP(selCallbackAddr, offCallbackAddr)
}
```

```
void DPMI_FreeCallback(FARPROC pfCallbackAddr)
{
WORD selCallbackAddr,offCallbackAddr;
selCallbackAddr=SELECTOROF(pfCallbackAddr);
offCallbackAddr=OFFSETOF(pfCallbackAddr);

_asm
{
mov  ax,DPMI_FREE_RM_CALLBACK
mov  bl,vector
mov  cx, offCallbackAddr
mov  dx, selCallbackAddr
int  31h
}
}

/* End  of  File */
```

列表 10.5　isr.c

```
#include <conio.h>
#include <windows.h>
#include "skeleton.h"
#include "uart.h"

#define MODE_REAL 1
#define MODE_PROT 2

void near C_CommonHandler(WORD mode)
{
BYTE ier,intid;
LPBYTE buf;
DEVICECONTEXT *hDevice;

HDevice=&Device1;
while(TRUE)
{
intid==_inp(hDevice→usIoBase+UART_REG_IIR);
if(intid==UART_IIR_NONE)
   break;
if(intid==UART_IIR_RXRDY)
{
```

```
    if((hDevice→RxBuf.In+1==hDevice→RxBuf.Out)
        ||(( hDevice→RxBuf.Out==0)&&
        (hDevice→RxBuf.In==hDevice→RxBuf.Size-1)))
    {
if(mode==MODE_PROT)
     PostMessage(hDevice→hwnd,WM_USER,0,NULL);
else
{
     hDevice→rmcs.ds=SELECTOROF((HDEVICE)hDevice);
     hDevice→rmcs.esi=OFFSETOF((HDEVICE)hDevice);
     (*hDevice→pfCallback)();
}
}
if(mode==MODE_PROT)
    buf=hDevice→RxBuf.ProtModeStart;
else
    buf=hDevice→RxBuf.RealModeStart;
buf[hDevice→RxBuf.In++]=_inp(hDevice→usIoBase+UART_REG_RDR);
if(hDevice→RxBuf.In>=hDevice→RxBuf.Size)
    hDevice→RxBuf.In=0;
}
else if(intid==UART_IIR_THRE)
{
if(hDevice→TxBuf.In==hDevice→TxBuf.Out)
{
ier=_inp(hDevice→usIoBase+UART_REG_IER);
ier&=~UART_IER_THRE;
_outp(hDevice→usIoBase+UART_REG_IER,ier);
}
else
{
if(mode==MODE_PROT)
    buf=hDevice→RxBuf.ProtModeStart;
else
buf=hDevice→RxBuf.RealModeStart;
_outp(hDevice→usIoBase+UART_REG_THR,buf[hDevice→TxBuf.Out++]);
if(hDevice→TxBuf.Out>=hDevice→TxBuf.Size)
    hDevice→TxBuf.Out=0;
}
}
}
if(hDevice→bFlags→bFlags & FLAGS_ON_SLAVE_PIC)
```

```
    _outp(SLAVE_PIC_STRL,EOI);
_outp(MASTER_PIC_STRL,EOI);
}
/* End of File */
```

列表 10.6　bimodal.inc

```
BufInfo STRUCT
Start          dd   0
BufSize        dw   0
IndexIn        dw   0
IndexOut       dw   0
BufInfo ENDS

DriverParams       STRUCT
ReadBufSize        dw    0
DriverParams       ENDS

RMCS  STRUCT
RM_EDI             dd   ?
RM_ESI             dd   ?
RM_EBP             dd   ?
RM_RES             dd   ?
RM_EBX             dd   ?
RM_EDX             dd   ?
RM_ECX             dd   ?
RM_EAX             dd   ?
RM_Flags           dw   ?
RM_ES              dw   ?
RM_DS              dw   ?
RM_FS              dw   ?
RM_GS              dw   ?
RM_IP              dw   ?
RM_CS              dw   ?
RM_SP              dw   ?
RM_SS              dw   ?
RMCS  ENDS

DeviceInfo  STRUCT
DevNum             dw   0
IoBase             dw   0
Irq                db   0
Flags              db   0
```

```
Hwnd             dw    0
OldProtHandler   dd    0
OldRealHandler   dd    0
RxBuf            db    SIZE BufInfo DUP(0)
TxBuf            db    SIZE BufInfo DUP(0)
numIntReal       dw    0
numIntProt       dw    0
Params           db    SIZE DriverParams DUP(0)
rmcs             db    SIZE RMCS DUP(0)
pfCallback       dd    0
DeviceInfo  ENDS
```

列表 10.7　bimodal.h

```
void far pascal ProtHandler(void);
void far pascal RealHandler(void);
void far pascal ProtModeCallback(void);
void far pascal SaveThisHandler(WORD this);
/* End  of  File */
```

列表 10.8　bimodal.asm

```
.286
INCLUDE device.inc
EXTRN _C_CommonHandler:NEAR

EXTRN AllocCSToDSAlias:FAR
EXTRN FreeSelector:FAR
EXTRN PostMessage:FAR

DATA SEGMENT WORD PUBLIC 'DATA'
DATA ENDS

_TEXT SEGMENT WORD PUBLIC 'CODE'

ASSUME cs:_TEXT

REALMODE  EQU  1
PROTMODE  EQU  2
WM_USER   EQU  0400h

public    RealModeDS
RealModeDS dw 0
```

```
RealHandler PROC FAR PASCAL PUBLIC
ASSUE ds:nothing
pusha
push  ds
mov   ax,cs:[RealModeDS]
mov   ds,ax
push  REALMODE
call  _C_CommonHandler
pop   ds
popa
iret
RealHandler  ENDP

ProtHandler PROC FAR PASCAL PUBLIC
ASSUE ds:nothing
pusha
push  ds
mov   ax,DATA
mov   ds,ax
push  PROTMODE
call  _C_CommonHandler
pop   ds
popa
iret
ProtHandler ENDP

ProtModeCallback PROC FAR PASCAL PUBLIC
push  ds
push  si
mov   ax.es:[di].RMCS.RM_DS
mov   ds,ax
mov   si,WORD PTR es:[di].RMCS.RM_ESI
cmp   [si].DeviceInfo.Hwnd,0
jz    no_post
mov   bx,WM_USER
xor   ax,a
push  ax
push  ax
push  ax
push  bx
push  [si].DeviceInfo.Hwnd
call  PostMessage
```

```
no_post:
pop   si
pop   ds
cld
lodsw
mov   es.[di].RMCS.RM_IP,ax
lodsw
mov   es.[di].RMCS.RM_cs,ax
lodsw
mov   es.[di].RMCS.RM_Flags,ax
add   WORD PTR es.[di].RMCS.RM_SP,4
iret
ProtModeCallback  ENDP

SaveThisHandler  PROC  FAR  PASCAL  PUBLIC
push  bp
mov   bp,sp
push  cs
call  AllocCSToDSAlias
push  es
mov   es,ax
mov   di,OFFSET RealModeDS
mov   ax,[bp+6]
stosw
push  es
call  FreeSelector
pop   es
mov   sp,bp
pop   bp
ret
SaveThisHandler  ENDS

_TEXT  ENDS
       END
: End  of  file.
```

列表 10.9　bimodal.def

```
LIBRARY      DRIVER
EXETYPE      WINDOWS
DATA         PRELOAD  FIXED  SINGLE
CODE         PRELOAD  FIXED  SHARED
```

列表 10.10　bimodal.mak

```
all: driver.dll

#DRIVER  DLL

#non-interrupt code(.C)
skeleton.obj:skeleton.c skeleton.h
  cl -c -W3 -ASw -Gsw2 -Zi -Oi $*.c

#interrupt code(.C)
isr.obj: isr.c skeleton.h
  cl -c -W3 -ASw -Gsw2 -Zi -Oi isr.c

#interrupt code(.ASM)
bimodal.obj: bimodal.asm bimodal.inc
  ml -c -W2 -Cp $*.asm

driver.dll: driver.def skeleton.obj isr.obj bimodal.obj
  link  skeleton+ bimodal, driver.dll, driver.map/CO /MAP, sdllcew libw
  /nod/noe, driver.def
  mapsym driver
  implib  driver.lib  driver.dll driver .def
  rc driver.dll
  copy  driver.dll\win31
```

第 11 章　编写使用 DMA 的驱动程序

前面几章所讲的所有驱动程序都依赖于微处理器向设备传送数据，以及从设备读取数据，或者对端口地址使用 in/out 操作，对内存地址使用 read/write 操作。有些设备读取数据的方式有所不同，如像磁盘控制器或者图形设备这样的能够处理大量数据的设备，通常采用直接内存访问（DMA）将数据传送到内存，从而减少数据总线的负载。

使用 DMA 的 Windows 驱动程序并不常见，因为 DMA 设备经常对物理地址直接寻址，要保证正确地对逻辑地址寻址是很困难的。本章将解释对一个 DMA 缓冲区的申请，并且说明怎样编写一个用 DMA 传送数据的驱动程序。

11.1　DMA 缓冲区的要求

用于 DMA 传送的缓冲区，不论是驱动程序到设备，还是设备到驱动程序，都必须遵守几个严格的要求。DMA 缓冲区必须是：

1. 物理上毗邻的；
2. 固定的和页面锁定的；
3. 定位在 64KB 的边界上。

这些要求是必须的，因为 DMA 控制器没有选择器或页的概念，并且不执行地址转换。控制器编程从一个物理地址开始，并且将每个字节传送后的地址加 1 或者减 1，用于产生下一个物理地址。

11.1.1　物理上毗邻

缓冲区空间物理上必须是毗邻的，因为处理器将线性地址空间划分为一系列 4KB 的页。通过处理器的页表，每个 4KB 的页都能够在物理空间中映射到不同的位置。而由映射到不同物理地址位置的页组成缓冲区。对于 DMA 来说不能工作，因为 DMA 控制器只能通过一系列的加（或者减）物理地址来计算。

11.1.2　固定的和页面锁定

要想明白固定的和页面锁定的要求，就要对图 11-1 进行分析。驱动程序使用所需缓冲区的物理地址对 DMA 控制器进行初始化，并通知控制器开始传送。于是控制器就传送一个字节，而且处理器恢复对总线的控制。Windows 的内存管理器接着决定移动一个段（在传送中使用过的），将段中的内容拷贝到另外的位置并且更新 LDT 中段的基地址。

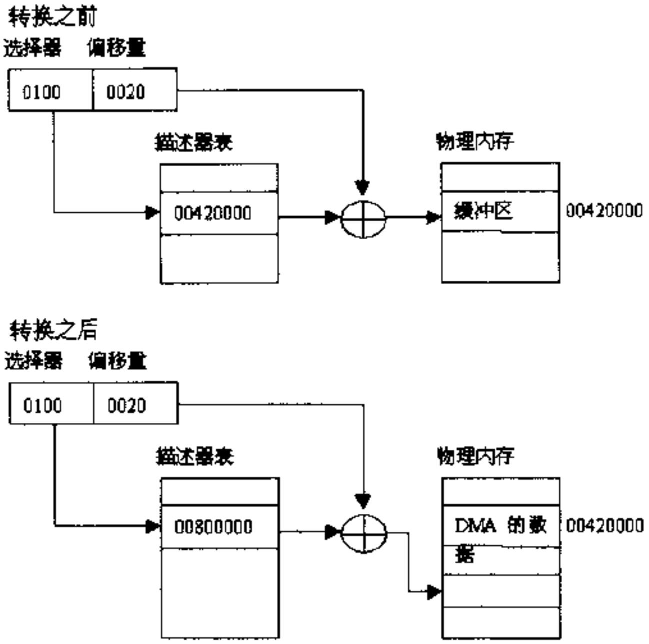

图 11-1　DMA 对固定内存的需求

以后，DMA 控制器又获得了总线，并且继续传送数据，不过它使用的还是在初始化过程中确定的原始地址。当传送结束时，驱动程序用选择器的值来检测新的数据，但是所期望的数据并不在选择器的新基地址所指示的位置。DMA 控制器将它存储在原始基地址所指示的位置。

我们可以通过分配固定的内存来防止这个问题，告诉 Windows 内存管理器不要移动这个段，段的改变由选择器的基址值所决定。但是仅有固定段的基准地址是不够的。内存管理器在处理页表以便重新映射组成段的 4KB 页的某些或者所有地址时，段的基准线性地址仍然保持固定。实际上，组成段的页中的某个页仍然可以被标注为“不存在的”，并被交换到硬盘。因此，DMA 的缓冲区在固定的同时还必须是页面锁定的。

11.1.3　定位在 64KB 的边界上

64KB 定位的要求是 PC 机的体系结构受到限定。最初的 PC 机使用的是 20 位物理地址，而 PC 机的 DMA 控制器芯片只有一个 16 位的地址寄存器。为了使 DMA 可以传送数据到 PC 机地址空间的整个 1MB 的任何位置，PC 系统的设计者们为 DMA 控制器扩充了一个页寄存器以存储地址的高 4 位。同时还增加了特殊的逻辑电路，以便这个页寄存器在 DMA 控制器将它的 16 位放到总线，而且将高 4 位也放到地址总线上，形成一个完整的 20 位地址。

当 PC-AT 出现时，这个页寄存器增加为 8 位，并且又加了特殊的逻辑电路，以便这个页寄存器在 DMA 控制器将它的 16 位放到总线，同时还要将这 8 位也放到地址总线上。

为了保持一致性，系统设计者们在微处理器使用 32 位地址总线时，仍然使用 24 位的 DMA 方案。

为什么会导致 64KB 界限的要求呢？下面来进行分析。假设将 1000H 字节用 DMA 传送到 6F000H 开始的物理地址去，其方法是：必须将低 16 位(F000H)写到 DMA 控制器的地址寄存器，将高 4 位(6H)写到相应的页寄存器中，还要将 DMA 控制器设置为传送 1000H 个字节，最后一个字节传送完后的物理地址是 70000H。但是传送完后的地址实际为 60000H。因为 DMA 控制器的地址寄存器正确地从 FFFFH 变为 0000H，页寄存器中的高 4 位并没有从 6H 增加到 7H。所以 DMA 的传送必须在同一个 64KB 的“页”上进行。

11.1.4 怎样分配 DMA 缓冲区

在第 4 章讲了怎样分配固定的和页面锁定的内存。所有 Windows 的 API 或 DPMI 调用都不能确定 64KB 的定位，不过这里有几个可用的方法。一个小的缓冲区很少突破 64KB 的界限，所以这种情况下的一个好的策略是保持分配（固定的和页面锁定的）缓冲区，直到得到一个合适的缓冲区位置，然后再将其他不用的缓冲区解除分配。

更大的缓冲区可能会跨越一个 64KB 的界限。为了获得这样的缓冲区，分配两次缓冲区以得到所需要的大小，然后使用不跨越 64KB 界限的那一半。

这样只剩下了最后一个要求——物理上毗邻的页，而且也没有 API 或 DMPI 调用来分配满足这个内存。另外还有一个问题，即使这样的缓冲区被分配了，驱动程序还要获得它的物理地址以便确定 DMA 控制器的基准地址。没有 API 或者 DPMI 的调用能得到一个物理地址，能够操作的就是通过使用 GetSelectorBase（）得到线性地址。

11.2 Windows 下 DMA 的 DOS 应用程序

有一点非常有趣，DOS 的应用程序在 Windows 下可以很好地进行 DMA 传送，而 Windows 应用程序要完成同样的任务则需要克服以上两个障碍。为什么会这样呢？原因就在于虚拟 DMA（VDMAD）设备在现场后面操作。VDMAD 存在的主要目的是为了使 DOS 应用程序能够像在 DOS 下执行 DMA 一样，即使保护模式内存转换比较困难。

VDMAD 捕获所有对 DMA 控制器的访问，并通过在内部截获那些数据不让其发向控制器。VDMAD 对控制器的基准地址寄存器很感兴趣。由于 DOS 应用程序运行在 V86 模式中，所以 DOS 应用程序写入这个寄存器中的地址是线性地址，而非物理地址。VDMAD 将这个线性地址转换成物理地址，再将这个物理地址写到控制器的地址寄存器中。另外，VDMAD 还将页面锁定成整个缓冲区，并且验证它是物理毗邻的。

如果它不是物理毗邻的，VDMAD 就用自身缓冲区替代物理地址，以符合所有 DMA 的要求。此时，VDMAD 通知控制器开始传送。当传送结束时，如果 VDMAD 自身的缓冲区被使用了，VDMAD 就将数据从自己的缓冲区拷贝到 DOS 应用程序的初始地址。不论怎样使用，DOS 应用程序所见到的只是一般的 DMA 传送，除非由于双重分配缓冲区而

使传送变得更慢。

11.3 DMA 的 Windows 应用程序可使用这个知识

因为VDMAD在Windows应用程序中就像在DOS应用程序中一样捕获访问，所以Windows应用程序可以使用CDMAD相互作用的知识来克服上面所讲的毗邻的页和物理地址问题。原则上，Windows应用程序的操作步骤如下：

1. 分配一个缓冲区。
2. 获得它的线性地址。
3. 将这个线性地址写入 DMA 控制器。
4. 依靠 VDMAD 来正确地完成每一件事。

这里唯一的技巧就是 Windows 应用程序并不使用缓冲区。DMA 基准地址寄存器是 24 位的。不论是使用 GlobalAlloc 分配的缓冲区，还是使用驱动程序数据段中静态分配的缓冲区，都位于线性地址空间的 2GB 以上，远远大于 24 位。正确的方法是使用 GlobalDosAlloc 来分配缓冲区，它保证线性地址位于 1MB 以下，并且适合 24 位的要求。

11.3.1 使用 VDS 缓冲区更好

上面所讲的方法很容易实现，但是有一个代价：GlobalDosAlloc（）将线性内存放在 1MB 以下。另一个解决方法是使用 VDMAD 自己的 DMA 缓冲区，只在传送的时间内使用，然后将数据拷贝到驱动程序的缓冲区。因为 VDMAD 输出对 Windows 应用程序的服务，这种借用 VDMAD 的缓冲区是可能的。这个 API 被称为虚拟 DMA 服务（VDS），可以通过 INT 4BH 得到。

VDS 的接口在表 11-1 中列出。总之，VDS 的函数包括页面锁定缓冲区、申请使用 VDS 缓冲区以及在 VDS 缓冲区和其他缓冲区间拷贝数据。

表 11-1 虚拟 DMA 服务

所有的服务		
INT 4BH，AH=81H，AL=Function，DX=Flags，ES:DI→DMA Descriptor		
DMA 描述表		
DWORD	Size;	
DWORD	Offset;	
DWORD	Segment/Selector;	
DWORD	Physical Address;	
名 称	函 数	说 明
Get Version	02H	返回 DMA 缓冲区的大小，无论内存是否是毗邻的
Lock DMA Region	03H	页面锁定缓冲区；如果缓冲区没有准备好，就借用 VDS 的缓冲区，并返回它的物理地址

续表

名　称	函　数	说　明
Unlock DMA Region	04H	页面解锁缓冲区
Scatter/Gather Lock	05H	页面锁定多个区域
Scatter/ Gather Unlock	06H	页面解锁多个区域
Request DMA Buffer	07H	借用 VDS 的缓冲区用于 DMA
Release DMA Buffer	08H	将 VDS 缓冲区返回给 VDS
Copy Into DMA Buffer	09H	将数据拷贝到 VDS 缓冲区
Copy Out Of DMA Buffer	0AH	将数据从 VDS 缓冲区进行拷贝
Disable DMA Translation	0BH	通知 VDS 编入控制器的地址是物理的，而不是线性的
Enable DMA Translation	0CH	通知 VDS 编入控制器的地址是线性的，而不是物理的

为了借用 VDS 缓冲区进行 DMA 传送，驱动程序调用 Request DMA Buffer 以返回 VDS 缓冲区的物理地址。在将这个地址写入控制器基准地址寄存器前，驱动程序调用 Disable DMA Translation。它通知 VDS，传送的地址不是线性地址，而已经变为物理地址了。

当传送结束时，驱动程序调用 Copy into DMA Buffer 或者 Copy Out Of DMA Buffer 来将数据移动到它自己的缓冲区。驱动程序还使用 Enable Translation 来再次允许传送，并使用 Release DMA Buffer 来释放缓冲区。

驱动程序分配固定的和页面锁定的缓冲区可以是物理毗邻的。如果这样，那么一直需要 VDS 缓冲区的驱动程序就会因为不必要地执行额外拷贝而带来损失。

另一个不需要更复杂地编程的选择是使用 Lock DMA Region 代替 Request DMA Buffer。Lock DMA Region 包含几个有用的函数。驱动程序传入一个缓冲区指针，VDS 首先检测缓冲区是否是 64KB 定位的和物理毗邻的。如果是，VDS 就锁定缓冲区内的所有页，并返回缓冲区的地址，得到的缓冲区满足所有 DMA 的要求；如果缓冲区不满足 DMA 的要求，VDS 就返回自身 DMA 缓冲区的物理地址。

驱动程序都调用 Disable DMA Translation，将物理地址写入控制器，并且开始传送。当传送完成时，驱动程序就调用 Enable DMA Translation（如果 VDS 的缓冲区被锁定了，还调用 Release Buffer）。调用 Copy into DMA Buffer 或者 Copy Out Of DMA Buffer 不是

必须的，因为如果使用了 VDS 缓冲区，VDS 会自动进行拷贝。如果是驱动程序自己的缓冲区用于传送，就不需要拷贝了。

使用 VDS 缓冲区进行 DMA 传送并不是理想的解决方法。对于所有使用 DMA 的 Windows 和 DOS 应用程序来说，必须共用一个 VDS/VDMAD 缓冲区，所以驱动程序在变为有效前不得不等待。更重要的原因是这个中介缓冲区导致额外的拷贝操作。

11.3.2 最佳的解决方案——助手 VxD

最佳的解决方案是使用一个“助手”VxD 来分配缓冲区。VxD 可以访问 VMM 的 API，而这个 API 可以分配物理毗邻的页。VxD 提供一个将缓冲区地址给驱动程序的 API，驱动程序将地址编入 DMA 控制器。这一节讲述一个助手 VxD DMABUF，以及一个测试 VxD 的 API 简单应用程序。

DMABUF（见列表 11.1 和 11.2）是一个只有两个基本部分的简单的 VxD。消息处理器（第一部分）只处理 Sys_Critical_Init 消息，缓冲区在响应这个消息时被分配。第二部分是 PM/V86 的 API，它包括获得缓冲区和释放缓冲区的函数。

物理毗邻的页只能在初始化时由 VxD 分配。Sys_Critical_Init 调用 VMM 的 PageAllocate（）来分配物理毗邻的缓冲区，定位缓冲区大小为 64KB。表 11-2 给出了 PageAllocate（）的参数。

表 11-2 PageAllocate 的参数

PageAllocate(nPages，pType，VMHandle，AlignMask，MinPhys，maxPhys，PhysAddrPTR，flags)	
参数	说明
nPages	4KB 页的数量
pType	PG_VM(专门对于 VM) PG_SYS(对所有的 VM 都有效) PG_HOOKED(专门对于使用 HookPageFault 的 VM)
VMHandle	VM 的处理；如果是 PG_SYS，就为 0
AlignMask	如果 Flags 中的 PageAlign 位被设置了，就使用 0FH 指定位于 64KB，1FH 指定位于 128KB
minPhys	允许的最小的物理页
maxPhys	允许的最大的物理页
PhysAddrPTR	指向物理地址将要返回的 DWORD 指针
Flags	PageZeroInit=01H PageUseAlign=02H PageContig=04H PageFixed=08H Page_locked=80H

DMABUF 将 nFlags 设置为(Bufsize/4096)，将 pType 设置为 PG_SYS，并将 VMHandle 设置为 0。为了满足 DMA 缓冲区的要求，必须将 AlignMask 的值设为 0FH 来迫使定位

64KB；将 minPhys 设为 0，将 maxPhys 设为 100 来迫使缓冲区位于 1MB 以下。1MB 的要求是必须的，因为 DMABUF 将它的服务输出到 DOS 应用程序以及 Windows 应用程序。

DMABUF 在 flags 参数中设置 PageUseAlign，PageContig 和 PageFixed。ALIGNMASK 参数迫使物理邻接和定位。PageFixed 分配在线性内存中固定、不会被交换出去的页。最后，PhysAddrPTR 参数是变量 BufferPhysical 的地址，在这里 VMM 将存放被分配缓冲区的物理地址。

从 PageAllocate（）返回时，DMABUF 就有了一个已分配的缓冲区以及关于它的几个地址。PhysAddrPTR 指向缓冲区的物理地址，而缓冲区的线性地址在 EDX 中。DMABUF 将物理量地址和线性地址分别存储在 BufferPhysical 和 BufferLinear 中，以便提供给以后的客户应用程序使用。VMM 还在 EAX 中返回缓冲区的 VMM 处理，但是 VxD 不使用这个值。

这个助手 VxD 只提供一个 API，所以它并不处理除 Sys_Critical_Init 以外的其他消息。对于其他的消息，设备控制过程会清除 carry 标志，然后返回。

11.4　DMABUF 的 API

DMABUF 对于保护模式和 v86 模式的 API 使用同一个入口点：DMABUF_Api。DMABUF_Api 检测客户 AX 寄存器中的函数号，并且如果函数号不被支持，就将错误返回到调用者。另外，DMABUF_Api 使用一个跳转表和在 EAX 中的函数号来调用 Get_Buffer 和 Release_Bufer。

Get_Buffer 命令将 DMABUF 的缓冲区标注为“在使用中”，并返回缓冲区的物理地址和线性地址。Release_Buffer 命令将 DMABUF 的缓冲区标注为“没有使用”。如果第二个应用程序在调用 Release_Buffer 前调用了 Get_Buffer，那么第二个对 Get_Buffer 的调用就会通过设置 carry 来返回失败。

下面将进一步说明 Get_Buffer。在进入 VxD 的 API 程序入口时，应用程序得到的寄存器值存储在 VM 的 CRS（客户寄存器结构）中，它由 EBP 指示。Get_Buffer 检测是否有错误，缓冲区是否已经被占用，或者 Client_CX 中的缓冲区长度的参数是否太大。这两种条件如有一个满足，Get_Buffer 将 Carry 标志设置为指示错误后返回。

如果一切都正确，Get_Buffer 在 DX:AX 中返回 DMA 缓冲区的物理地址，在 BX:CX 中返回线性地址。用户应用程序将物理地址编入 DMA 控制器，并且使用线性地址和 SetSelectorBase（）创建映射到缓冲区的选择器。

程序代码如下所示：

```
mov  eax, BufferPhysAddr      ; EAX=32-bit physical address
mov  [ebp, Client_AX], ax     ; store low word
shr  eax, 16                  ; ax=high word
mov  [ebp. Client_DX], ax     ; store nigh word
mov  eax, BufferLinear        ; EAX=32-bit linear address
```

```
mov  [ebp, Client_BX], ax      ; store low word
shr  eax, 16                   ; ax=high word
mov  [ebp, Client_CX], ax      ; store high word
```

11.5 使用助手 VxD——USEAPI.C

一个非常简单的应用程序 USEAPI.C（见列表 11.3 和 11.4）说明了怎样使用助手 VxD。这个应用程序既可以编译成 Windows 应用程序，又可以（使用#define）编译成 DOS 应用程序。DOS 应用程序使用 printf（）进行显示，而 Windows 应用程序恰在 WinMain（）中调用 MessageBox（），所以这里没有消息循环。

获取 VxD 入口点以及通过这个入口点调用 VxD 的 Get_Buffer 服务程序代码对于 Windows 和 DOS 应用程序是一样的。对于 Windows，需要调用 AllocSelector（），SetSelectorBase（）和 SetSelectorLimit（）；而对于 DOS，只需要将消息地址右移 4 位，因为微处理器会将它左移 4 位以形成物理地址。

应用程序显示 DMA 缓冲区的物理地址、它的线性地址以及指向缓冲区的指针，还增加了通过指针写到缓冲区的程序代码，使得应用程序实际执行到设备的 DMA。最后，应用程序调用 DMABUF 的 Release_Buffer 释放缓冲区，以便别的应用程序可以使用它。

11.6 总　　结

在其他操作系统下，当 DMA 传送提供重要的执行优势时，在 Windows 下不会看到同样的优势，除非你决心编写助手 VxD。如果你不能够实现毗邻、固定和页面锁定的限定条件，DMA 传送将在 DMA 传送后招致额外的拷贝，付出比 DMA 传送更多的代价。

如果提供助手 VxD，DMA 就可以改善执行过程。编写助手 VxD 并不难，但是要记住，用户需要修改 Windows 的配置，以便 Windows 能够加载和初始化 VxD。

列表 11.1　dmabuf.asm(DMA 的 VDS 接口)

```
PAGE   58, 132
TITLE  dambuf.asm
; ================================================
       .386p
; ************************************************
;            INCLUDE
; ************************************************
.XLIST
       include   VMM.INC
       include   VPICD.INC
       include   DEBUG.INC
```

```
.LIST
;************************************************
;     VIRTUAL  DEVICE  DECLARATION
;************************************************
Declear_Virtual_Device  DMABUF, 1, 0, DMABUF_Control, 0DB0h, \
        Undefined_Init_Order, DMABUF_Api, DMABUF_Api
;************************************************
; Inittialization  data

VxD_IDATA_SEG

VxD_IDATA_ENDS

;************************************************
; Global  data

VxD_DATA_SEG

BufferSize          EQU  (64*1024)-1
NumPages            EQU  BufferSize/4096
BufferPhysical      dd   0
BufferLinear        dd   0
BufferOwner         dd   0
PTE                 dd   0

DMABUF_Call_Table  dd   OFFSET32 Get_Buffer
                  dd   OFFSET32 Release_Buffer
DMABUF_MAX_API      EQU  ($-DMABUF_Call_Table)/4

VxD_DATA_ENDS

VxD_LOCKED_DATA_SEG

VxD_LOCKED_DATA_ENDS

;************************************************
; Initialization  code

VxD_ICODE_SEG
;************************************************
;  DMABUF_Sys_Critical_Init
;  DESCRIPTION:
```

```
;  ENTRY:     EBX is SYS VM handle
;  EXIT:
;  USES:
; ==============================================================

BeginProc  DMABUF_Sys_Critical_Init
      mov        edi, OFFSET32 BufferPhysical
      VxDcall    _PageAllocate, <NumPages, PG_SYS, 0, 0Fh, 0, 100h, edi, \
                 <PageUseAligbn+PageContig+PageFixed>>
      Jnz         Alloc_ok
      Debug_Out   'Failed VPICD_Virtualize_IRQ'
      Stc
      Ret

Alloc_ok:
      Mov         [BufferLinear], edx
      Clc
      Ret
EndProc DMABUF_Sys_Critical_Init

VxD_ICODE_ENDS

; ***********************************************************

VxD_CODE_SEG

; ***********************************************************
;  DMABUF_Control
;  DESCRIPTION:dispatch control messages to correct handlers
;  ENTRY:
;  EXIT:   Carry clear if no error
;  USES:
; ==============================================================

BeginProc  DMABUF_Control
       Control_Dispatch  Sys_Critical_Init, DMABUF_ Sys_Critical_Init
       Clc
       Ret
EndProc  DMABUF_Control

; ***********************************************************
;  DMABUF_Api
```

```
; DESCRIPTION:
; ENTRY: EBX=current VM handle
;        EBP→Client Register Structure
; EXIT:
; USES:
; ================================================

BeginProc DMABUF_Api
Movzx  eax, [ebp, Client_AX]
Cmp   eax, DMABUF_MAX_API
Jea   bad_api
Call  DMA_Call_Table[eax*4]
Ret

Bda_api
Debug_Out 'DMABUF invalid function #eax'
Err_api
Mov   [ebp, Client_EAX], 0BADDBADDh
Or    [ebp, Client_Eflags], CF_Mask
Ret
EndProc DMABUF_Api

BeginProc Get_Buffer
Cmp   [BufferOwner], 0
Jne   SHORT err_api
Cmp   [ebp.client_cx], BufferSize
Ja    err_api

mov  eax, BufferPhysAddr
mov  [ebp, Client_AX], ax
shr  eax, 16
mov  [ebp, Client_DX], ax
mov  eax, BufferLinear
mov  [ebp, Client_BX], ax
shr  eax, 16
mov  [ebp, Client_CX], ax
mov   [BufferOwner], ebx
ret
EndProc Get_Buffer

BeginProc Release_Buffer
mov   [BufferOwner], 0
```

```
ret
EndProc  Release _Buffer

VxD_CODE_ENDS

END

; End  of  File
```

列表 11.2　dmabuf.def

```
LIBRARY         DMABUF
DESCRIPTION      'Win386  DMABUF  Device(Version1.0) '
EXETYPE         DEV386
SEGMENTS
                _LTEXT  PRELOAD  NONDISCARDABLE
                _LDATA  PRELOAD  NONDISCARDABLE
                _ITEXT  CLASS  'ICODE'  DISCARDABLE
                _IDATA  CLASS  'ICODE'  DISCARDABLE
                _TEXT   CLASS  'PCODE'  NONDISCARDABLE
                _DATA   CLASS  'PCODE'  NONDISCARDABLE
EXPORTS
                DMABUF_DDB   @1
```

列表 11.3　useapi.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef DOS
#include <dos.h>
#endif
#include <windows.h>

typedef void (far *PVOIDFN)(voi)

#deifne DMABUF_ID 0xDB0
#deifne DMABUFFUNC_GETBUFFER 0
#deifne DMABUFFUNC_FREEBUFFER 1

static char  buf[1024]={0};

PVOIDFN GetVxDApiEntry(int VxDId)
```

```
{
PVOIDFN pfApi:
_asm
{
xor  di, di
mov  es, di
mov  bx, VxDId
mov  ax, 1684h
int  2fh
mov  WORD PTR pfApi+2, es
mov  WORD PTR pfApi, di
}
return (pfApi):
}

#ifdef DOS
main()
#else
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
            LPSTR lpCmdLine, int nCmdShow)
#endif
{
PVOIDFN pfDmaBufApi;
char far *DmaBufPtr;
unsigned short DmaBufSize, usSel, myds;
unsigned long DmaBufPhys;
unsigned long DmaBufLin;
char tmpbuf[128]
DmaBufSize=32*1024;
pfDmaBufApi=GetVxDApiEntry(DMABUF_ID);
if(pfDmaBufApi)
{
    _asm
{
mov  cx,  DmaBufSize
mov  ax,  DMABUFFUNC_GETBUFFER
call DWORD PTR pfDmaBufApi
mov  WORD PTR  DmaBufPhys+2, dx
mov  WORD PTR  DmaBufPhys, ax
mov  WORD PTR  DmaBufLin+2, bx
mov  WORD PTR  DmaBufLin, cx
}
```

```
#ifdef DOS
printf("Physical=%08lx\nLinear=%08lx\n, DmaBufPhys, DmaBufLin);
DmaBufPtr=(char far *)( DmaBufLin>>4);
printf(" Pointer-%Fp\n", DmaBufPtr);
printf("Now accessing DmaBufPtr\n");
*DmaBufPtr='X';
printf ("Freeing buffer);
#else
wprintf(tmpbuf, "Physical=%08lx\nLinear=%08lx\n, DmaBufPhys,
DmaBufLin);
strcat(buf, tmpbuf);
_asm mov myds, ds
usSel=AllocSeletor(myds);
SetSelectorBase(usSel, DmaBufLin);
SetSelectorLimit(usSel, DmaBufSize);
DmaBufPtr=MAKELP(usSel, 0);
wprintf(tmpbuf, "Pointer=%08lx\n", DmaBufPtr);
strcat(buf, tmpbuf);
strcat(bug, " Now accessing DmaBufPtr\n");
MessageBox(NULL, buf, "USEAPI", MB_OK);
*DmaBufPtr='d';
FreeSelector(usSel);
MessageBox(NULL, "Freeing buffer, "USEAPI", MB_OK);
#endif
{
mov ax, DMABUF_FUNC_FREEBUFFER
call DWORD PTR pfDmaBufApi
}
return 0
}
}
/*End of File*/
```

列表 11.4 useapi.def

```
LIBRARY         USEAPI
EXETYPE         WINDOWS
CODE            PRELOAD MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE
HEAPSIZE        4096
STACKSIZE       8192
EXPORTS
```

列表 11.5 makefile

```
INCLUDE-$(INCLUDE); \karen\tools\include

.all: dmabuf.386 useapi.exe

#VXD

dmabuf.obj:dmabuf.asm
  masm5 -DDEBUG dmabuf.asm

dmabuf.386:dmabuf.obj
  link386 dmabuf.obj, dmabuf.386, dmabuf.map/MAP, , dmabuf.def;
  mapsym32 dmabuf
  addhdr dmabuf.386

#TEST APPLICATION
useapi.obj useapi.c
  cl -W3 -c -AS -Gsw2 -Zi -Oi $*.c

useapi.exe::useapi.def useapi.obj
  link useapi, useapi, useapi.map/MAP/CO, slibcew libw/nod, useapi.def;
  rc useapi.exe
```

第 12 章　使用实模式服务

Windows 应用程序在保护模式中运行，可以使用实模式下的 DOS 和 BIOS 服务。无论是在 Windows 下运行的 DOS 应用程序调用 C 库中的读函数，还是 Windows 应用程序调用 Windows 的 lread 函数，这些读操作最终都归结为调用 DOS 的 INT 21H，就像是程序在 DOS 下运行一样。为了使用 DOS 和 BIOS 服务，Windows 应用程序还可能使用其他（如 TSR，DOS 设备驱动程序等）的实模式服务。

为了从保护模式应用程序中正确执行实模式代码，编程人员必须克服许多障碍。首先最重要的，微处理器必须从保护模式切换到 V86 模式或实模式，然后再执行反方向的操作。寻址以及其他方面的不同导致许多更微妙的障碍。保护模式的数据可能位于 1MB 以上，但它对于实模式程序来说是访问不到的，这使得阐述的传递变得复杂了。另外，在微处理器切换回保护模式时，一些通过实模式代码在段寄存器中返回的参数将导致一些异常情况，因为它们对于选择器是无效的。

Windows 应用程序的编程人员不需要关心以上的问题。通过 INT 21H 的调用以及对以上做一些问题处理工作，Windows 内核可以很容易使 Windows 应用程序使用 DOS 和 BIOS 服务。设备驱动程序不会导致一个被保护的原形。

Windows 应用程序和 DOS 一样，从不使用实模式服务，Windows 驱动程序可能使用一个已存在的提供硬件设备支持的实模式 TSR 或 DOS 服务驱动程序。在这种情况下，开发人员需要很好地理解上面提到的转换问题，因为 Windows 的内核对它不知道的 TSR 和设备驱动程序不提供相同级别的转换支持。另外，还有一些 Windows 不支持 INT 21H 服务（见表 12-1），所以驱动程序开发人员在使用这些不支持的服务时，必须提供转换。

表 12-1　Windows 不支持的 DOS 函数

函　数	说　明	函　数	说　明
INT 20H	结束程序	INT 21H	Func 24H 获取相关记录
INT 25H	绝对磁盘读	INT 21H	Func 25H 获取中断向量
INT 26H	绝对磁盘写	INT 21H	Func 27H 随机块读
INT 27H	结束和保持驻留	INT 21H	Func 28H 随机块写
INT 21H	Func 00H 结束过程	INT 21H	Func 35H 设置中断向量
INT 21H	Func 0FH 用 FCB 打开文件	INT 21H	Func 38H 获取本地页
INT 21H	Func 10H 用 FCB 关闭文件	INT 21H	Func 44H　Subfunc 02
INT 21H	Func 14H 连续读	INT 21H	Func 44H　Subfunc 03
INT 21H	Func 15H 连续写	INT 21H	Func 44H　Subfunc 04

续表

函 数	说 明	函 数	说 明
INT 21H	Func 16H 用 FCB 创建文件	INT 21H	Func 44H Subfunc 05
INT 21H	Func 21H 随机读	INT 21H	Func 65H 获取扩展的本地信息
INT 21H	Func 22H 随机写	INT 21H	Func 67H 设置处理记数
INT 21H	Func 23H 获取文件长度		

Windows 对其中的一些工作提供帮助，Windows 在应用程序发生软件中断时会根据需要自动切换微处理器的模式。然而，Windows 对于不知道的设备不会自动提供缓冲区转换，因为它根本不知道哪个寄存器保护指针。如果 Windows 驱动程序需要用实模式服务交换指针，它就必须自己作一些转换工作。

下一节将探讨几个调用实模式服务和将参数传送到实模式服务的技术。

12.1 关于 DOS 设备驱动程序

根据定义，一个 DOS 设备驱动程序存在一个特殊的接口，应用程序把它当作一个文件进行访问，使用 DOS 的 Open，Read，Write，Close 调用它。应用程序可以通过 INT 21H 直接实现这些 DOS 调用，也可以通过使用 C 运行库的低级文件函数_open（）和_read（）等进行调用。除了这些标准调用之外，许多的 DOS 设备驱动程序通过 IOCTL（I/O 控制）命令支持扩展接口。C 运行时设备驱动程序的 IOCTL 命令不提供支持，如果需要产生 IOCTL，必须通过嵌入汇编语言来产生 DOS 的 IOCTL。

Windows 应用程序通过同样的文件接口访问 DOS 设备驱动程序，Windows 内核捕获 INT 21H，并且为 DOS 设备驱动程序访问提供一些必须的转换。例如 Windows 在 read 和 write 调用中查看传递给 DOS 设备驱动程序的缓冲区地址，并且检测这个缓冲区是否位于 1MB 以上。如果是，这个缓冲区就不能使用，因为 DOS 设备驱动程序在实模式中执行，不能访问 1MB 以上的任何东西。所以 Windows 在调用 DOS 设备驱动程序前取代自身缓冲区的地址。另外；即使 Windows 应用程序的缓冲区恰好在 1MB 以下，Windows 应用程序编入的缓冲区地址也是保护模式的地址，而不是 DOS 设备驱动程序可使用的实模式地址。所以 Windows 必须将保护保护模式指针转换为实模式指针。

除了像 Open，Read，Write，Close 这样的文件函数，许多 DOS 设备驱动程序通过 DOS 的 IOCTL 函数调用（INT21H，AH=44H）还支持设备指定的函数性。一个专门的子函数是通过 AL 寄存器指定的。例如 IOCTL 的子函数包括接收控制数据（AL=2）和写控制数据（AL=3），这两个子函数以及其他一些子函数只使用一个在 DS：DX 中传递的缓冲区参数。

由于每个子程序使用的缓冲参数由 IOCTL 接口定义，所以 Windows 为所有的 IOCTL 缓冲区自动执行转换。不过，设备驱动程序不对它所选择的缓冲区内容进行中断。所以设

备驱动程序可以将接收控制数据缓冲区作为一个数据结构，并且这个结构包括指针。此时 Windows 不知道缓冲区包含的需要转换的指针。这说明通过 IOCTL 调用驱动程序的读控制数据的 Windows 应用程序必须处理这些嵌入指针的转换。

12.1.1 对 IOCTL 的专门处理

下面仔细分析一下 Windows 驱动程序在读控制数据 IOCTL 调用 DOS 设备驱动程序时需要做什么，这些 DOS 设备驱动程序需要处理嵌入指针的转换。在这个例子里，通过 DS:DX 传送到设备驱动程序的缓冲区并不是一个字符缓冲区，而是一个如图 12-1 所示的 CONTROL_DATA 结构。CONTROL_DATA 包含一个指向整数数组的指针。设备驱动程序将用表列所支持的速度来填充由应用程序分配的数组。

```
typedef struct
{
void far *speeds;
int numspeeds;
} CONTROL_DATA;
```

图 12-1 设备驱动程序使用的数据缓冲区的读控制

在 1MB 以下分配 CONTROL_DATA 结构并没有严格的要求，因为由 Windows 提供的自动缓冲区转换知道在 DS：DX 中的指针。但是 Windows 不知道 CONTROL_DATA 包含指针，所以 Windows 驱动程序必须保证速度数组在 1MB 以下，并且速度指针是一个实模式指针。

已经存在的 Windows API 函数 GlobalDosAlloc ( )专门提供实模式缓冲区，并且永远分配在线性地址空间的 1MB 以下。GlobalDosAlloc ( )有一个 DWORD 类型的返回值，它包括一个在保护模式下对缓冲区寻址的选择器和一个在实模式下对缓冲区寻址的段。这两部分在 Windows 驱动程序中都会用到，因为 Windows 驱动程序需要一个保护模式的指针来对常规缓冲区寻址以及一个实模式指针指到设备驱动程序。其中指针的偏移量都为 0。

LOWORD 宏选取选择器，而 MAKELP 将其转换成远程指针。Windows 驱动程序使用这个指针来初始化缓冲区，并且在设备驱动程序填满它后访问它。实模式段由 HIWORD 宏提取，并且使用 MAKELP 将其转换成实模式指针。注意不要废弃这个实模式指针，虽然它在保护模式下是无效的。

图 12-2 中的代码使用了上面的方法，从一个 DOS 设备驱动程序中获取设备的信息。在速度数组被设备驱动程序填满以后，代码就扫描数组以决定设备是否支持高速率。

Windows 只能对 DOS 设备驱动程序提供自动转换，因为它知道这些驱动程序可以通过 INT 21H 访问。即使驱动程序是通过 INT 21H 访问，Windows 只能在确切知道所需参数时才能提供转换。Windows 知道它所需要确定的读和写的每件事，但是却不知道它是 IOCTL。

```
BOOL SupportsHighSpeed(void)
{
CONTROL_DATA cdata;
BOOL highspeed=0;
WORD far *speeds;
WORD numspeeds=8
DWORD dw;
WORD i;
#define IOCTL_READ_CONTROL_DATA  2
dw=GlobalDosAlloc(numspeeds*sizeof(WORD));
cdata.numspeeds=numspeeds;
cdata.speeds=MAKELP(HIWORD(dw), 0);
speeds=MAKELP(LOWORD(dw), 0);
DosIoctl(IOCTL_READ_CONTROL_DATA, &cdata, sizeof(cdata));
for(i=0; i<numspeeds; i++)
{
    if(*speeds++>9600)
        return (TRUE);
}
```

图 12-2　当发生 DOS IOCTL 时使用 GlobalDosAlloc ( )

TSR 又有所不同，因为每个 TSR 有它自己的接口和参数传递方法。为此，连接 TSR 的 Windows 代码要比连接 DOS 设备驱动程序的代码做更多的工作。下一节将对此进行解释。

12.1.2　介绍 DOS TSR

DOS TSR 几乎都是通过软件中断来访问的，它们的参数通过寄存器来传递。有时参数也通过堆栈传递，这时通过软件中断的初始化调用返回一个或者几个函数的地址，而应用程序稍后直接调用这些地址，在堆栈中传递参数。

使用在寄存器中传递参数的中断方法更简单一些，下面首先介绍它。为了保证软件中断，Windows 应用程序使用与 DOS 程序一样的方法，使用运行库 int86 ( )或者汇编语言。

Windows 捕获所有的软件中断指令，并且首先决定软件中断是否被“支持”，就像 DOS 和 BIOS 一样需要专门的转换处理。表 12.1 列出了不支持的中断。如果不需要特殊的处理，Windows 就只将处理器切换到实模式下，并且在 IVT 中调用软件中断。当实模式软件中断遇到 IRET 指令时，Windows 就将处理器切换回保护模式，并且继续执行 Windows 应用程序。

如果 Windows 驱动程序不需要向 TSR 传递指针，TSR 也不向驱动程序返回指针，此时所要做的就是初始化处理器的寄存器并执行软件中断。如果需要交换指针，那么驱动程序就必须做一些特别的工作。

12.1.3 通过缓冲区传递数据

一个 Windows 驱动程序在调用 DOS TSR 以及通过缓冲区交换数据时，必须采取一些特殊的预防措施。如果 Windows 驱动程序提供缓冲区并且 TSR 填充了它，预防措施就包括分配实模式地址内存以及向 TSR 传递缓冲区地址。如果 TSR 拥有自己的缓冲区，并且将缓冲区地址交给了 Windows 驱动程序，那么预防措施就是正确地形成指针去访问实模式缓冲区。其方法如下：

1. DOS 应用程序“拥有”缓冲区

通过寄存器由 TSR 返回的指针是实模式指针，Windows 驱动程序不能直接使用它。作为保护模式应用程序，Windows 应用程序必须使用保护模式指针。技巧就是将实模式的指针转换成保护模式的指针。

实模式指针包括段和偏移量，而保护模式指针包括选择器和偏移量。段和选择器的区别在于处理器怎样将它们变换成物理地址。在实模式中，处理器通过变换一个简单的算术运算来计算物理地址：

物理地址＝段<<4＋偏移量

在保护模式中，选择器和物理地址之间没有算术联系。相反，每个选择器都有一个相关的基准地址（存储在一个由操作系统维护的表中），处理器通过将偏移量与这个基准地址相加来获得物理地址。

将实模式指针转换成保护模式指针是这样实现的：驱动程序创建保护模式选择器，其基准地址等于实模式指针所操作的物理地址。Windows 提供以下一系列的 API 函数来实现这一功能：AllocSeletor ( )，SetSelectorBase ( )以及 SetSelectorLimit ( )。图 12-3 给出了怎样调用这几个函数的例子。

```
void far *RealPtrToProtPtr(WORD seg , WORD off)
{
char far *ptrProt;
WORD myDs , sel;

_asm mov myDs , ds
sel=AllocSelector(myDs);

//Set Selector's Linear address to (seg<<4)+offset
base=(seg<<4)+off
SetSelectorBase(sel, base);

//Set Selector limit to 64K
SetSelectorLimit(sel, 64K);

Return(MAKELP(SEL, 0));
}
```

图 12-3 将实模式指针转换为保护模式指针的函数

2．Windwos 应用程序拥有的缓冲区

现在看看数据怎样从 Windows 传递到 TSR。这里的问题和前面 IOCTL 中遇到的一样。从 Windows 驱动程序传递到 TSR 的缓冲区必须位于 1MB 以下，因为在实模式中执行 TSR 时，处理器只能寻址 1MB 以下的内存。所以要使用 GlobalDosAlloc ( )来分配这样的缓冲区，并且返回的 DWORD 类型的参数用来审查保护模式指针和实模式指针。

一旦缓冲区被分配，驱动程序必须将缓冲区的地址传递给 TSR。一般来说，TSR 希望从寄存器传递参数，而不是堆栈，所以 Windows 应用程序必须将实模式段和 0 偏移量调入到 TSR 需要的寄存器对中。DOS 应用程序调用 TSR 有两种方法，一是使用汇编语言直接加载处理器的寄存器并且必须保证软件中断，另外一种是使用 C 库函数中的 INTDOSX ( )函数。Windows 使用一个缓冲区来调用 TSR，方法并不是非常简单，它需要使用一个新的 WINDOWS 的 API 。

3．使用 DPMI 调用 TSR

Windwos 的设计者知道应用程序与驱动程序通信的重要性，所以要提供一系列的服务来实现它。这些服务不是 Windwos 中普通的 API 函数，而是由 Windows 通过 INT 31H 提供的 DPMI 接口。调用 TSR 并将它传递给缓冲区指针的服务是 DPMI 的仿真实模式中断（见图 12-4），仿真实模式中断通过“实模式调用结构”传递寄存器信息。图 12-5 给出了它的 C 语言描述。

```
AX=0300h
BL=interrupt  number
BH=flags
Bit 0=1 to rest interrupt controller and A20 line
CX=number of word copied from prot mode statck to real mode stack
ES:DI=far pointer to real mode call structure
```

图 12-4　调用 DPMI 的仿真实模式中断的参数

```
struct{
unsigned long edi;
unsigned long esi;
unsigned long ebp;
unsigned long resl;
unsigned long edx;
unsigned long ecx;
unsigned long ebx;
unsigned long eax;
unsigned shorts flags;
unsigned shorts es;
unsigned shorts ds;
unsigned shorts fs;
```

```
unsigned shorts gs;
unsigned shorts ip;
unsigned shorts cs;
unsigned shorts sp
unsigned shorts ss;
} REAL_MODE_CALL_STRUC;
```

图 12-5　DPMI 的仿真实模式中断使用的“实模式调用结构”

驱动程序根据 TSR 的需要填充这些寄存器。接着驱动程序使用嵌入的汇编来填充处理器的寄存器，并且执行 INT 31H。软件中断不使用堆栈参数，所以 CX 等于 0。

当 Windows 的驱动程序执行 INT 31H 时，DPMI 首先将“实模式调用结构”的内容拷贝到一个在实模式中可以访问的中介区。然后 DPMI 切换到实模式，并将“实模式调用结构”的数据拷贝到相应的处理器寄存器。最后 DPMI 执行软件中断。

TSR 运行时，察觉不到它是被保护模式的应用程序调用。TSR 返回时，DPMI 切换回保护模式，并从中介区将“实模式调用结构”拷贝回 Winodws 应用程序的初始结构。TSR 返回后，驱动程序使用先前用 MAKELP 创建的指针和保护模式选择器来检测缓冲区的内容。

图 12-6 的例子说明怎样将缓冲区传递给 TSR。其中 TSR 通过 INT 14H 调用，并在 ES:BX 中传递缓冲区的指针。

```
DWORD dw;
WORD seg, sel;
char far *buf;
READL_MODE_CALL_STRUC RmCallStruc;

dw=GlobalDosAlloc(256);
seg=HIWORD(dw);
RmCallStruc.es=seg;
RmCallStruc.bx=0;
_asm{
mov  ax, 0300h;
mov  bl, 14h;
xor  bh, bh;
xor  cx, cx;
mov  es, SEG RmCallStruc
mov  di, OFF RmCallStruc
INT  31h
}
buf=MAKELP(LOWORD(dw), 0);
//buffer can now acess data filled in by TSR
```

图 12-6　向 TSR 传递缓冲区

4. 通过地址调用 TSR

调用实模式 TSR 不常用的方法是通过固定的地址。图 12-7 给出的程序代码是通过软件中断 50H 获得实模式函数地址，然后使用 DPMI 的调用实模式过程来调用函数。另外，

```
//TRS expects functioncode in AX
REAL-MODE-CALL-STRUC rmCALLStruc;
DWORD DW;
WORD seg;
Move rmCALLStruc.eax=7h;          //Function:Get entry point
CALL DPMI Simulate Real Mode interrupt throygh int 31h
Asm
{
move ax, 0300h                    //DPMI function
move bl, 50h                      //int number
move bh, 0                        //flags
xor cx, cx                        //stack words
move es, SEG rmCALLStruc
move di, OFF rmCALLStruc
int 31h
}
//TSR returned the entry point adress in CX:DX
//Move it to CS:IP of call structure
rmCALLStruc.cs=(WORD) rmCALLStruc.ecx;
rmCALLStruc.ip=(WORD) rmCALLStruc.edx;
//Allocate memory for buffer to give to TRS
dw=GlobalDosAlloc(sizeof(MyStruc));
seg=LOWORD(dw);

//CALL TSR entrypoint, whose function prototype is:
//void pascal TsrEntry(WORD FunctionCode, char far *pBuffer)
asm
{
Use DPMI CallReal Mode Procedure with Far Ruturn Frame
Move ax, 0301h                    //DPMI function
Xor bh, bh                        //flags
Move cx, 2                        //stack words
Move es, SEG rmCALLStruc
Move DI, OFF rmCALLStruc
Int 31h
}
```

图 12-7 程序代码通过一个函数地址调用 TSR 的技术

有两个参数将传递给实模式函数，第一个是通知 TSR 的函数代码，第二个是指向缓冲区的指针。缓冲区必须是被 GlobalDosAlloc()分配的，并且传递给缓冲区的地址必须是由 GlobalDosAlloc()返回的实模式段。

12.1.4　TSR 调用 Windows 应用程序

Windows 应用程序和 TSR 之间的另一种交互是交给 TSR 一个回调地址，TSR 存储这个地址，并在稍后中断控制器的执行期间调用此功能。通常 TSR 是通过寄存器将必要的信息传递给回调函数。为此，需要使用 DPMI 的分配实模式回调地址过程（INT 31H，AX=03H）。

然而，Windows 应用程序只给 TSR 一个实模式指针是不够的，因为 Windows 是一个多任务环境，TSR 在执行回调时必须确保正确的 VM。

12.2　增强模式中的回调

为了理解在增强模式中使用回调的难度，现在需要再对虚拟机（VM）进行说明。在增强模式下有两种 TSR：一种是句柄 TSR，它映射到单个 VM 的地址空间，并且在创建新的 DOS VM 以及从命令行加载 TSR 时被创建；另一个 TSR 就是全局 TSR，它在 Windows 开始前被加载。全局 TSR 映射到所有 VM 的地址空间，在内存中只有一个全局 TSR 的拷贝，但是每个 VM 都有一个从物理地址映射到这个拷贝的线性地址。

TSR 通过捕获中断向量来完成它的工作。如果一个全局 TSR 还被应用程序回调，那么它必须知道哪个 VM 被知道。假设 TSR 的中断处理器使用 VM1 的地址空间，而回调在 VM2 的地址空间。那么回调就捕获工作，因为回调地址只在 VM2 运行时才可以获得，而不是 VM1。TSR 将调用执行无用单元地址的指针，系统就可能会因此崩溃。

解决回调的方法：

最简单的解决方法是建立 Windows 知道的 TSR。Windows 知道的 TSR 不直接使用回调，而是一直等到 SYSVM 成为当前的 VM。TSR 可以通过 VMM 提供的 INT 2fH 接口使 SYSVM 被调度，而在 SYSVM 是当前的 VM 时，VMM 就回调到 TSR。这时 TSR 可以安全使用 DPMI 的回调。最后由于保护模式中的处理器和正确的 VM，使 Windows 应用程序中的回调可以执行。

这样，当 Windows 应用程序通过回调和 TSR 通信时，通信包括以下步骤：

对于 Windows 应用程序：

1. 使用 DPMI“分配实模式回调地址”来分配一个回调。
2. 将 DPMI 的回调地址传递给 TSR。

对于中断时的 TSR：

3. 如果当前 VM 是 SYSVM，就转到步骤 5。
4. 如果当前 VM 不是 SYSVM，就使用 INT 2FH 来切换 VM 和回调到 SYSVM。

如果当前 VM 是 SYSVM:

5．调用 DPMI 的回调地址，DPMI 切换到保护模式并调用 Windwos 应用程序。

对于 Windows 应用程序：

6．回调在保护模式中执行（在 SYSVM 中），并通过实模式调用结构访问 TSR 数据。

7．回调调整实模式调用结构 CS:IP，并返回。

8．DPMI 切换回实模式，并返回 TSR。

12.3 编 程 细 节

Windows 驱动程序将受到两方面的影响，一是 TSR 回调寄存器以及驱动程序本身的回调，二是 TSR 需要增加代码处理 VM 的切换。

图 12-8 说明了 Windows 驱动程序怎样使用 DPMI 的分配实模式回调地址。为了在 TSR 注册回调，Windwos 驱动程序首先需要获得实模式回调地址。通过 DPMI 的分配实模式回调地址，这个服务的输入参数指向被调用函数的一个保护模式指针以及指向"实模式数据结构"的保护模式指针。DPMI 返回子函数的段和偏移量。现在可以将返回的段和偏移量传递给 TSR。

```
void far *pffoo;
WORD CallbackSeg;
WORD CallbackSeg;
pFoo=&Foo;
_asm
{
mov  ax, 0303h
mov  es, SEG RmCallStruc
mov  di, OFFSET RmCallStruc
mov  si, pFoo
push ds
mov  ds, pFoo+2
int  31h
pop  ds
mov  CallbackSeg, cx
mov  CallbackOff, dx
}
//Give CallbackSeg and CallbackOff to TSR as callback
```

图 12-8　使用 DPMI 的分配实模式回调地址

在使用回调地址前，TSR 必须检查当前的 VM。如果是 SYSVM，TSR 通过回调地址并返回；如果是另外的 VM，TSR 就使用 INT 2FH 服务，切换 VMS 和回调。图 12-9 列出

了这个服务所需的参数。为了使用这个服务，必须告诉 WMM/INT 2FH 想调度的 VM，要给 VM 的优先权，以及 VM 发生时的地址。由于程序要执行的代码是 Windows 的驱动程序，所以 TSR 需要调度 VM1，它就是 SYSVM，而优先权参数一般为 0。最后给 VMM/INT 2FH 回调地址，与 Windows 驱动器给 TSR 一样。

```
INT 2FH
AX=1685H (Function code Switch VMs and Callback)
BX=switch to this VM(id)
ES:DI=address of function to call when Vmis current
CX=flags
PEF_Wait_For_STI(0001h) to wait until interrupts are enabled
PEF_Not_Crit(0002h) to wait until critical section is unowned
DS:SI=priority boost
Cur_Run_VM_Boost (00000004h) to run the VM for its full time slice
Low_Pri_Device_Boost (00000010h) to give the VM moderate
  proority over other VMs
High_Pri_Device_Boost (00001000h) to give the VM
  significant proority over other VMs
Critical_Section_Boost (00100000h) to give the VM same
  proority as if in critical section
Time_Critical__Boost (00400000h) to give the VM Higher
  proority than a critical section
```

图 12-9 切换 VM 和回调的参数

TSR 在调用 INT 2FH 后返回，VMM 将调度 SYSVM。SYSVM 实际上已成为当前的 VM，接着 VMM 调用注册的回调函数地址。这时，回调就会在 Windwos 驱动程序中最终执行（即在保护模式和正确的 VM 中）。

图 12-10 中的程序说明怎样在 TSR 中使用切换 VM 和回调。

最后，让我们来检测一下 SYSVM 是当前 VM 时的运行情况。当 TSR 使用的回调最终运行时，这个回调实际是 DPMI 的子程序。这个子程序有几个重要的任务。它将所有实模式寄存器的值拷贝到一个调用结构，然后切换到保护模式。接着用指向调用结构的指针加载 ES:DI，用寻址实模式堆栈的指针加载 DS:SI。最后在 DPMI 的“分配实模式回调地址”注册的地址处调用 Windows 驱动程序。

现在驱动程序开始执行。由于驱动程序运行于中断环境，所以没有 DOS 或 BIOS 函数，而只有所列中断安全的 Windows API 函数可以使用。在实模式调用结构中的指针是实模式指针，只有将其转换为保护模式指针后才能使用。

驱动程序可以通过检测调用结构中相应的字段来访问从 TSR 中传出的寄存器的值，还可以修改调用结构中送到 TSR 的寄存器的值。如果 TSR 与驱动程序的通信是通过堆栈参数而非寄存器参数，那么驱动程序可以用 DS:SI 中的保护模式指针访问实模式堆栈。

```
mov  ax, 1683h ; function Get current VM
int  2fh
cmp  CallbackVM, 1 ; is SYSVM current
jz   same_VM

mov  ax, 1685 ; function switch VM and Callback
mov  bx, 1 ; switch  to  SYSVM
mov  cx, 0 ; flags
mov  si, 0 ; priority boost

; cakk-back address registered by Windows driver goes in ES:DI
mov  di, WORD PTR CallbackAddr+2
mov  es, di
mov  di,  WORD PTR CallbackAddr
int  2fh
jmp  xit

same_VM:
pushf  ; SYSVM is current , use callback directly
call DWORD PTR:CallbackAddr

xit:
```

图 12-10　TSR 使用切换 VM 和回调

图 12-11 中的程序说明怎样解决调用结构中的 CS:IP，它必须在离开 Windows 驱动程序的回调之前立即被执行。

```
Callback
; do your own thing
; access real mode call structure via ES:DI
if parameters needs from real mode statck
; use DS:SI
call DoYouOwnThing
; extrack proper real mode CS and IP from
; top of real mode stack, pointed to by DS; SI.
; Put CS and IP values into real mode call structure
cld
loasw
mov WORD PTR ds:[di.RM_IP], ax ;  real mode IP
mov WORD PTR ds:[di.RM_CS], ax ;  real mode CS
add WORD PTR ds:[di.RM_SP], 4  ;  toss old CS:IP from stack
iret
```

图 12-11　在 Windows 驱动程序的回调中固定实模式的 CS 和 IP

12.4 总 结

DPMI 的服务使 Windows 驱动程序与 DOS 的 TSR 以及设备驱动程序的通信变得实用。这样就可以只开发一个驱动程序来满足 DOS 应用程序和 Windows 应用程序的需要。如果曾经有一个 DOS 驱动程序，那么将它修改为 Windows 可用的驱动程序，可能是最简单的开发途径。

不过，如果从头开始编写一个基于 DOS 的驱动程序，通过本章会发现，那些通过软件中断调用和期望在寄存器中传递所有参数的驱动程序最容易执行和被支持。

第 13 章　编写标准模式的驱动程序

由于 Windows 的实模式已经过时，目前大都不再使用，所以现在的开发者们需要的是支持标准模式和增强模式的驱动程序。现在大部分用户运行增强模式，但是仍然有一些用户运行于标准模式，并且开发者们必须开发标准模式的驱动程序。本章将介绍如何在驱动程序中支持标准模式。

13.1 基本概念

13.1.1 硬件的端口映射

在增强模式中，程序依靠 IOPM 执行输入输出指令。标准模式的 Windows 根本不控制 I/O 访问，运行在标准模式 Windows 下的进程能够访问任意的 I/O 空间，这与增强模式有明显的不同，在增强模式下 VxD 能够捕获 I/O 对一个设备的访问。

13.1.2 内存映射的硬件

在很多情况下，标准模式访问内存映射的硬件与在增强模式中类似，图 13-1 给出了一个可以在标准模式和保护模式中创建指针并工作的程序。

```
void far *MapPhysToPtr(DWORD PhysBase, DWORD PhysSize)
{
WORD myDs, sel;
WORD HiBase, LoBase;
_asm mov myDs, ds
sel=AlloSelector(myDs);
SetSelectorLimit(sel, physSize);
_asm
{
    mov cx, PhysBase
    mov bx, PhysBase+2
    mov di, Physsize
    mov si, PhysSize+2
    mov ax, 0800h
    int 31h
    mov HiBase, bx
```

```
    mov  LoBase, cx
}
SetSelectorBase(sel, MAKELONG(LoBase, HiBasae));

Return(MAKELP(sel, 0));
}
```

图 13-1　在标准模式和保护模式中创建指向内存映射设备的指针

13.1.3　标准模式中的选择器

在标准模式中，选择器不能寻址大于 64KB 以上的空间，因为描述表只有 16 位宽。要访问大于 64KB 的设备就需要多个选择器。标准模式的这种多个选择器的类型被定义为 Tiled Selector。

标准模式的程序可以使用巨型指针来访问大于 64KB 的区域，也可以在 Windows 的驱动程序中通过使用巨型指针达到同样的效果，在形成指针时更加注意就可以了。Windows 选择器不是通常的算术关联，这样在一个选择器上加 1 就会导致另一个有效的选择器。不过即使选择器是有效的，它也不会映射到物理空间的下一个 64KB。

然而，程序可以使用 DPMI 来分配算术关联的选择器。选择器间并不是相差 1，而是一个被称为 AHINCR 的神奇数字。如果要分配几个寄存器并使它们在内存中是连续的，可以创建出与实模式中相同的结果。图 13-2 所示的 MapWithTiledSelectors ( )说明了这个技术。

```
#include  <stdio.h>
#include  <string.h>
#include  "windows.h>
#include  "dpmi.h"
static char buf[1024]={0};
WORD MapWithTiledSeletors(DWORD
ulRegionBase, DWORD ulRegionSize)
{
WORD usSel, usNumSel, usFirstSel;
#define ANHINC 8
usNumsel= ulRegionSize/(64L*1024L);
if(ulRegionSize%(64L*1024L))
    usNumsel++;
DpmiAllocateLdtDescrs(usNumsel, & usFirstSel);
usSel= usFirstSel;
ulRegionSize--;
while(usNumSel--)
```

```
{
DpmiSetSegmentBase(usSel, ulRegionBase);
DpmiSetSegmentLimit(usSel, ulRegionSize);
UlRegionBase+=0x00010000;
UlRegionSoze+=0x00010000;
UsSel+=AHINC;
}

return (usFirstSel);
}
int PASCAL WinMain(HANDLE hInstance, HANDLE hPreInstance, LPSTR
          lpCmdLine, int nCmdShow)
{
DWORD ulbase, ulSize;
BYTE huge *pBuf, c;
WORD usSel;
long j;
ulBase=0x00F00000;
ulSize=0x00020000;
usSel=MapWithTiledSelectors(ulBase, ulSize);
pBuf=MAKELP(usSel, 0);
wsprintf(buf, "Now accessint region
%08lx(size %08lx)\n, ulBase, ulSize);
MessageBox(NULL, buf, "USEAPI", MB_OK);
For(j=0; j<ulSize; j++) c=*pBuf++;
DpmiFreeLdtDescrs(usSel);
Return 0;
}
```

图 13-2　使用巨型指针访问内存

13.2　标准模式中的中断

第 4 章中讲过的控制中断的技术可以应用于标准模式中。实际上，第 4 章中的例子驱动程序同时适用于增强模式和标准模式，不过必须注意增强模式与标准模式中断的调度机制的区别。

主要的区别在于增强模式包括虚拟机管理器 VxD 以及虚拟机，而标准模式中没有。即使如此，标准模式中的中断控制与增强模式中的--样复杂。

原因之一是标准模式在任何机器上都可以运行，而它们在结构上明显不同。在应用程序层，这一区别察觉不到，但对于底层驱动程序的开发确非常重要。

Windows 的标准模式在保护模式和 V86 模式（而非实模式）间切换运行 Windows 的

应用程序和 DOS/BIOS 服务。如果这两个模式运行时发生中断，处理器自动切换到保护模式，并使用中断描述器表来获得中断控制器的地址。在标准模式中，中断控制器位于 Windows 的核心。核心控制器确定中断发生时处理器的模式，然后决定是从中断向量表调用 V86 模式的控制器，还是从中断描述器表中调用保护模式控制器。

如果中断发生在实模式或 v86 模式中，则运行在 IVT 中的实模式或 V86 模式控制器；如果在保护模式中发生中断，则执行 IDT 中的保护模式控制器。

由于处理器在保护模式和实模式之间经常切换，所以 Windows 就要同时保护一个 IDT 和一个 IVT，而且 Windows 必须保证在模式切换时使用正确。在 Windows 的中断控制器安装时，IVT 和 IDT 有时两个都被更新，有时只有 IVT，有时只有 IDT，这依赖与安装中断控制器的方法。也就是说，要看中断是发生在哪个模式中。

13.3　从保护模式和实模式中获取中断

在 Windows 开始前只有一个中断表，即实模式 IVT。所有的控制器由位于 BIOS，DOS 和 TSR 中的 IVT 所指示，如图 13-3 所示。

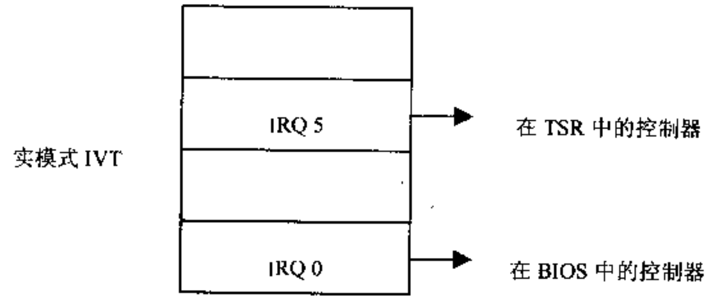

图 13-3　在 Windows 运行前只有一个 IVT

在初始化保证模式 Windows 时，Windows 的 DOSX 组件创建一个保护模式 IDT。它的每个入口都是一个被称为 PM-to-RM-反射器的控制器。这个处理器首先切换到实模式，然后调用实模式 IVT 中的相应控制器。例如 IRQ0 的 PM-to-RM-反射器 切换到实模式，并调用实模式 IVT 中的 IRQ0 控制器，如图 13-4 所示。

Windows 在安装中断控制器时修改这两个表或者其中的一个。Windows 应用程序主要使用 DOS 的设置向量服务（INT 21H AH=25H）来安装控制器。例如，假设 Windows 驱动程序为 IRQ5 安装控制器，在安装时 Windows 就调用 INT 21H 来调用设置向量服务，于是 Windows 就在保护模式 IDT 中，它能够直接得到在保护模式中产生的 IRQ5。不过 Windows 也在实模式 IVT 中安装了 PM-to-RM-reflector。如图 13-5 所示。这样，如果在实模式中发生了中断 IRQ5，这个 PM-to-RM-reflector 就会切换到保护模式并调用 IDT 中的控制器。

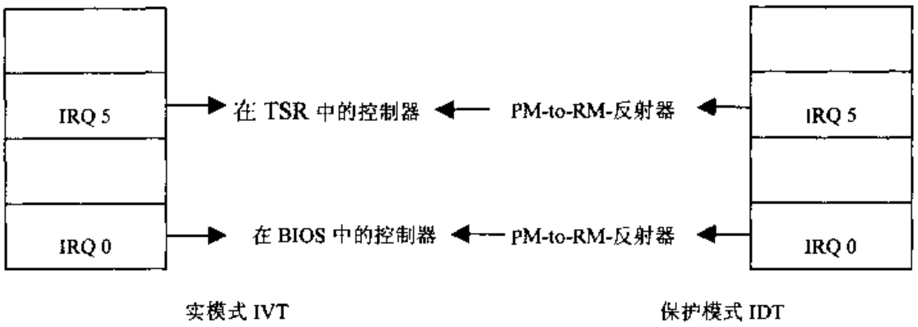

图 13-4　Windows 初始化后的 IVT 和 IDT

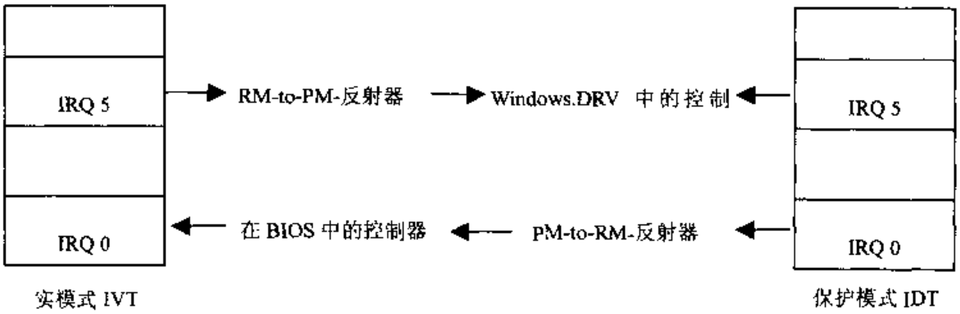

图 13-5　Windows 使用 DOS 的 SetVector 更新 IVT 和 IDT

13.4　分离实模式控制器和保护模式控制器

一些驱动程序同时安装保护模式控制器和实模式控制器。例如 Windows 系统串口驱动器（COMM.DRV）。COMM.DRV 通过 DOS 设置向量来安装保护模式，通过 DPMI 实模式设置向量（INT 31H，AH=25H）来安装实模式控制器。DPMI 将串口控制器直接装入实模式 IDT，并改写由所谓 DOS 设置向量安装的 RM-to-PM-反射器，如图 13-6 所示。

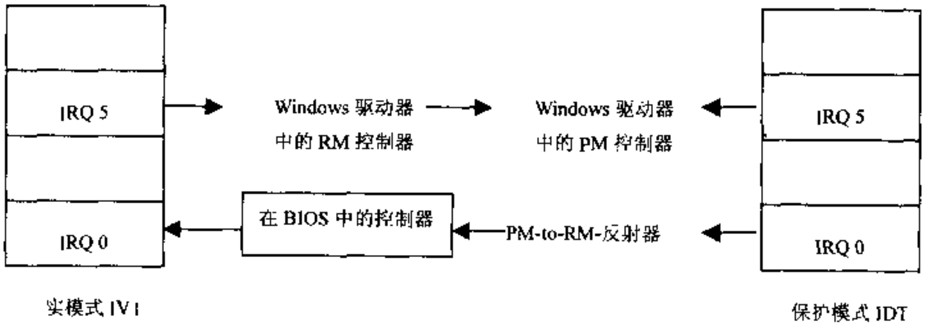

图 13-6　Windows 驱动器使用 DPMI 安装分离的实模式和保护模式控制器

SYSTEM.DRV 提供一个更复杂的中断控制方案。这个驱动程序控制硬件的定时器中断 IRQ0。当加载 SYSTEM.DRV 时，它首先使用 DOS 的获取向量来检索当前的 IRQ0，然后用 DOS 的设置向量在 IDT 中安装自己的控制器。现在保护模式的 IRQ0 中断直接进入 SYSTEM.DRV 的控制器中，而实模式的 IRQ0 中断提供 RM-to-PM-反射器反射到 SYSTEM.DRV 。

不寻常的是 SYSTEM.DRV 在保护模式中服务于定时器中断后，它必须调用原始的实模式控制器将中断传递到 Windows 开始前控制向量的实模式 TSR。为此，SYSTEM.DRV 只会调用原始的保护模式 IRQ0 的控制器。那么怎样通过调用保护模式控制器来实现调用原始的实模式控制器呢？

实际上，SYSTEM.DRV 调用的控制器是 DOSX 的 PM-to-RM-反射器。但是 PM-to-RM-反射器 的代码不能使用实模式的 IVT 来获取原始实模式控制器的地址，因为现在的实模式 IVT 保护的是 RM-to-PM-反射器，如图 13-7 所示。为此，Windows 在初始化期间建立了一个实模式 IVT 的拷贝，SYSTEM.DRV 可以用它来找到原始的实模式控制器，如图 13-8 所示。

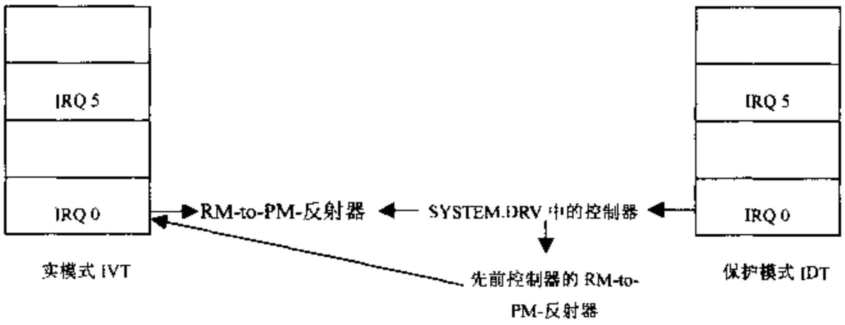

图 13-7　如果 PM-to-RM-反射器使用 IVT 将导致无限循环

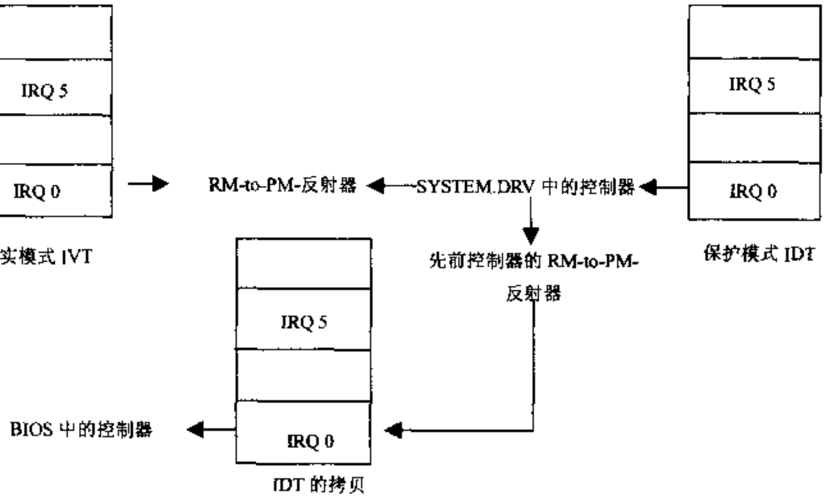

图 13-8　PM-to-RM 反射器必须使用 IVT 调用 BIOS 控制器

13.5 总　　结

今天，许多驱动程序开发者容易忽视标准模式 Windows。不要担心，只要设计仔细，就可能开发出一个适用于两种模式的程序。虽然保护模式和标准模式在一些重要方面有所不同，但许多常用的操作都是相同的，如内存 I/O、设备 I/O 以及简单的中断等。只要注意了这些，就可以编写出在以上两种模式中都可以工作的程序来。

MapWithTiledSelectors ( )函数首先分配了几个选择器，每户 64KB 一个，使用 DPMIAllocate LDT Descriptors(见表 13.1 和表 13.2)。表 13.3 是生成文件，表 13.4 是 maptiled 文件的定义。

列表 13.1　dpmi.h

```
#include <windows.h>

#define SELECTOR_CODE 0x001A
#define SELECTOR_DATA 0x0012

WORD DpmiAllocateLdtDescrs(WORD usNumDesc, WORD *pFirstSel);
WORD DpmiGetNextSelectorInc(WORD *pIncVal);
WORD DpmiFreeLdtDescrs(WORD usFirstSel);
WORD DpmiGetSegmentBase(WORD usSel, DWORD *pulBase);
WORD DpmiSetSegmentBase(WORD usSel, DWORD ulBase);
WORD DpmiSetSegmentLimit(WORD usSel, DWORD ulLimit);
WORD DpmiSetDescrAccessRights(WORD usSel, WORD usRights);
WORD DpmiGetDescr(WORD usSel, LPBYTE pDescr);
/* End of file */
```

列表 13.2　dpmi.c

```
#include "dpmi.h"

#deifne DPMI_FUNC_ALLOC_LDT_DESC      0X0000
#deifne DPMI_FUNC_FREE_LDT_DESC       0X0001
#deifne DPMI_FUNC_GET_NEXT_SEL_INC    0X0003
#deifne DPMI_FUNC_GET_SEGMENT_BASE    0X0006
#deifne DPMI_FUNC_SET_SEGMENT_BASE    0X0007
#deifne DPMI_FUNC_SET_SEGMENT_LINIT   0X0008
#deifne DPMI_FUNC_SET_DESCR_RIGHTS    0X0009
#deifne DPMI_FUNC_GET_DESCR           0X000B
```

```
WORD  DpmiAllocateLdtDescrs(WORD usNumDesc, WORD *pFirstSel)
{
WORD usRetVal=0;
WORD usLocal;

_asm
{
mov  ax,  DPMI_FUNC_ALLOC_LDT_DESC
mov  cx, usNumDesc
int  31h
jnc  xit
mov  usRetVal, 1
xit:
mov  usLocal, ax
}
*pFirstSel=usLocal;
return  usRetVal;
}

WORD  DpmiFreeLdtDescrs(WORD usFirstSel)
{
WORD  usRetVal=0;

_asm
{
mov  ax,  DPMI_FUNC_FREE_LDT_DESC
mov  bx, usFirstSel
int  31h
jnc  xit
mov  usRetVal, 1
xit:
}
return  usRetVal;
}

WORD  DpmiGetNextSelectorInc(WORD *pIncVal)
{
WORD usRetVal=0;
WORD usLocal;

_asm
{
```

```
mov  ax,  DPMI_FUNC_GET_NEXT_SEL_INC
int  31h
jnc  xit
mov  usRetVal, 1
xit:
mov  usLocal, ax
}
*pFirstSel=usLocal;
return  usRetVal;
}

WORD  DpmiGetSegmentBase(WORD usSel, DWORD *pulBase)
{
WORD usRetVal-0;
WORD usHi, usLo;

_asm
{
mov  ax,  DPMI_FUNC_GET_SEGMENT_BASE
mov  bx, usSel
int  31h
jnc  xit
mov  usRetVal, 1
xit:
mov  usHi, cx
mov  usLo, dx
}
*pFirstSel=(usHi<<8)+usLo;
return  usRetVal;
}

WORD  DpmiSetSegmentBase(WORD usSel, DWORD ulBase)
{
WORD usRetVal=0;
WORD usHi, usLo;

usHi=HIWORD(ulBase);
usLo=LOWORD(ulBase);

_asm
{
mov  ax,  DPMI_FUNC_SET_SEGMENT_BASE
```

```
mov  bx, usSel
mov  cx, usHi
mov  dx, usLo
int  31h
jnc  xit
mov  usRetVal, 1
xit:
}
return  usRetVal;
}

WORD  DpmiSetSegmentLimit(WORD usSel, DWORD ulLimit)
{
WORD usRetVal=0;
WORD usHi, usLo;

usHi=HIWORD(ulLimit);
usLo=LOWORD(ulLimit);

_asm
{
mov  ax,  DPMI_FUNC_SET_SEGMENT_LIMIT
mov  bx, usSel
mov  cx, usHi
mov  dx, usLo
int  31h
jnc  xit
mov  usRetVal, 1
xit:
}
return  usRetVal;
}

WORD  DpmiSetDescrAccessRights(WORD usSel, WORD usRights)
{
WORD usRetVal=0;

_asm
{
mov  ax,  DPMI_FUNC_SET_DESCR_RIGHTS
mov  bx, usSel
mov  cx, usRights
```

```
int 31h
jnc xit
mov usRetVal, 1
xit:
}
return usRetVal;
}

WORD DpmiGetDescr(WORD usSel, LPBYTE pDescr)
{
WORD usRetVal=0;

_asm
{
mov ax, DPMI_FUNC_GET_DESCR
mov bx, usSel
push es
les di, pDescr
int 31h
popes
jnc xit
mov usRetVal, 1
xit:
}
return usRetVal;
}
/* End of file */
```

列表 13.3　makefile

```
all: maptiled.exe

#TEST APPLICATION
maptiled.obj: maptiled.c
  cl -c -W3 -AS -Gsws -Zi -Od $*.c

dpmi.obj: dpmi.c
  cl -c -W3 -AS -Gsws -Zi -Od $*.c

maptiled.exe: maptiled.def maptiled.obj dpmi.obj
  link maptiled+dpmi, maptiled, maptiled.map/CO/MAP, slibcew libw/nod,
maptiled.def
  rc maptiled.exe
```

列表 13.4　maptiled.def

```
LIBRARY      USEAPI
EXETYPE      WINDOWS
CODE         PRELOAD  MOVEABLE  DISCARDABLE
DATA         PRELOAD  MOVEABLE
HEAPSIZE     4096
STACKSIZE    8192
EXPORTS
```

第 14 章　定时器和软件中断

本章主要讲解两个重要的问题。首先讲解怎样使用定时器来周期性执行服务子程序，然后讲解怎样创建一个控制软件中断的 VxD。

14.1　定　时　器

应用程序和驱动程序经常选用定时器来获得对处理器的周期性控制。在 DOS 系统下，获得周期性运行的唯一途径是截获定时器中断，它可能每 55 毫秒发生一次（18.2 次每秒）。在 Windows 下，使用定时器有几种不同的选择。下面我们就对这些方法进行介绍。

14.1.1　使用 SetTimer（）

许多 Windwos 编程人员对函数 SetTimer（)很熟悉。调用这个函数创建的定时器可以发出消息 WM_TIMER，或者在定时器期满时调用一个回调函数。定时器的周期是 55 毫秒。但实际上 SetTimer ( )对于需要"立即"通知的应用程序没有什么用处，原因是 SetTimer ( )与服务子程序间的通信使用的是消息系统而不是中断，在定时器期满和 WM_TIMER 消息的处理过程或回调函数之间可能会损失不少时间。Windows 使用中断定时器中断来实现这些定时，但是中断控制器所做的一切是设置定时器已经发生的标志指示，然后应用程序进入它的消息循环并调用 GetMessage ( )，这样从中断开始又有许多时间被损耗了。

使用 SetTimer（)执行回调函数会消耗同样多的延时，因为 Windows 定时器中断控制器不直接调用回调函数，而是控制器设置标志，稍后用 GetMessage ( )来检测这个标志。

14.1.2　调用 INT 1CH 和 INT 8H

要避免上面所讲的延时，Windows 驱动程序可以直接控制定时器中断。许多 DOS 应用程序调用软件中断控制器（INT 1CH），而非硬件中断控制器（INT 8H），因为在 BIOS 中的 INT 8H 控制器在处理定时器中断后产生 INT 1CH。但是 INT 1CH 在 Windows 下不能工作，硬件定时器中断的 Windows 控制器将中断送到 INT 8H 控制器。当 INT 8H 控制器调用 INT 1CH 控制器时，处理器处于实模式状态。所以调用 INT 1CH 的 Windows 驱动程序不会见到这个中断，因为它运行于保护模式中。

也就是说，Windows 驱动程序使用 INT 8H，Windows 在切换到实模式和调用 BIOS 的 INT 8H 控制器前，要首先调用所有的保护模式 INT 8H 控制器。但不幸的是，这个方法也存在一些硬件中断所有的限制：可用的 Windows API 函数只有 PostMessage ( )；通过

调用 PostMessage ( )，INT 8H 控制器只能执行临界时间的动作并退出其他操作的执行；Windows 过程在接到消息时会结束处理。

14.1.3　不要依赖于每秒 18.2 次

INT 8H 控制器的方法非常不完善，因为在 DLL 中 INT 8H 控制器并不能保证每 55 毫秒调用一次。在保证模式下，只有 Windows 在运行，INT 8H 控制器就每 55 毫秒执行一次。如果用户执行 DOS 应用程序，那么 Windows 应用程序和 DLL 甚至 Windows 自己就都被调到硬盘，而 Windows DLL 中的 INT 8H 控制器就不执行，直到退出 DOS 而返回 Windows。INT 8H 控制器接着会在下一个定时器周期被调用，但是这时距离上一次调用可能已经有几个小时了。

在增强模式下，实际的硬件定时器中断由 VxD，VDT（虚拟定时器设备）服务，接着 VTD 为 VM 仿真定时器中断。因为 VM 看到的仿真中断并非实际的，所以定时器中断的频率有所不同。VTD 给前台 VM 的定时器频率是每秒 18.2 次，而前台 VM 得到的只有每秒 3 到 4 次。也就是说，应用程序不能每 55 毫秒接收一次中断。

14.1.4　使用 timeSetEvent ( )

给 Windows 驱动程序最精确的周期性通知是由 Windows 的多媒体服务 timeSetEvent ( ) 提供的。它的时间可以精确到 1 毫秒，而且可以在标准模式和保护模式中提供同样的 API 和解决结果。

14.1.5　使用 VxD

上述所讲的方法如果都失败了，就使用 VxD。可以使用 VMM 的 Set_Global_time_Out ( )服务来迫使回调函数在几个毫秒后再执行，这就创造了一个“只有一次”的定时器。VxD 可以在回调中再次调用 Set_Global_time_Out ( )来开始下一个定时器，这样就提供了一个连续运行的定时器了。

通常这个方案的周期是 20 毫秒一次，不过 VxD 可以采取更好的方发，即通过调用 VTD_Begin_Min_Int_Period ( )来得到等于 1 毫秒的周期。如果分辨率不支持这个方法，它就会返回一个错误。当 VxD 结束它的定时工作时，必须调用 VTD_End_Min_Int_Period ( ) 来将定时器的频率恢复到初始值。

14.1.6　不要使用 GetTickCount ( )

Windows 提供了函数 GetTickCount ( )，不过这个函数是不准确的，因为 Windows 使用它自己的 INT 8H 控制器来保持定时器的发生。前面已经讲过，Windows 的 INT 8H 控制器不能每 55 毫秒调用一次。为了保证 55 毫秒周期，应用程序必须使用 BIOS 保持

的计数。

14.1.7 使用 BIOS 的周期计数或者是 timeGetTime ( )

Widows 驱动程序要定位当前的时间信息，应该使用 DOS 或者 BIOS 服务，而非 Windwos 的 GetTickCount ( )函数。要使用 BIOS 服务，因为 DOS 服务返回的时间形式不便于减少，它是通过时钟设备来获取 BIOS 的计数，所以 BIOS 获取的时间更好。

如果需要比 55 毫秒更精确，前面讲的多媒体定时器就可以提供当前的时间，而且精确到 1 毫秒。函数 timeGetTime ( )还可以以毫秒级返回 Windows 开始后的时间。

14.2 软 件 中 断

Windows 的应用程序不使用软件中断来调用其他的 Windows 应用程序，所以 Windows 编程者们很少写保护模式的软件中断控制程序。然而，Windows 的应用程序可能会安装一个连接初始控制器的软件中断控制器，以便监控对软件中断的调用。

14.2.1 连接到软件中断

在 Windows 下连接到软件中断的方法如下：首先得到当前控制器的地址并存储它，然后安装新的控制器；接着当新的控制器被调用时，通过将标志压入堆栈以及对保存的地址进行远程调用来调用初始的控制器。

可以使用 DOS 的设置向量来安装软件中断控制器，使用 Windows 更新保护模式的 IDT 来调用控制器。图 14-1 说明了怎样安装软件中断控制器。

```
void HookInt21h(DWORD NewHandler)
{
WORD flags;
DWORD ver;
extern WORD _0000h;
LPDWORD lpdwInt21Vector;
DWORD OldVector;

if((HIBYTE(ver)==3)&&(HIBYTE(ver)==0)
{
if(flags&WF_STANDARD)
{
    lpdwInt21Vector=(LPWORD)MAKELONG(0X0084, &_0000H);
    OldVector=* lpdwInt21Vector;
    * lpdwInt21Vector=NewVector
```

```
}
else if(flags&WF_ENHANCED)
{
    _asm
    {
    mov  ax, 0201h
    mov  bl, 21h
    mov  cx, SEGMENT NewHandler
    mov  dx, OFFSET NewHandler
    int  31h
    }
}
}
else
{
_asm
{
mov  ax, 25h
mov  bl 21h
push ds
mov  ds, SEGMENT NewHandler
mov  dx, OFFSET NewHandler
int  21h
pop  DS
    }
}
```

图 14-1　调用 INT 21H 来安装软件中断控制器

Windows 驱动程序安装的软件中断控制器可以从其他的 Windows 应用程序看到软件中断，但是不能从 DOS 的应用程序看。如果想从 DOS 应用程序看，就要使用 VxD。

14.2.1　用 VxD 捕获软件中断

列表 14.1 和 14.2 所列的例子 VxD(TRAPINT)说明 VxD 怎样在 V86 和保护模式中调用软件中断，它调用 INT 21H 并查看“删除文件”调用(AH=41H)。如果 Carry 位被设置为“删除文件”调用，TRAPINT 就立即返回，以防止删除；如果 AH 不是“删除文件”，TRAPINT 就连接到先前的控制器。

TRAPINT 在消息 Sys_Critical_Init 处理期间调用 INT 21H，在两种模式下的调用有所不同。在 V86 模式中，TRAPINT 调用 Hook_V86_Mode_Chain()服务，在 EAX 中传递中断号，在 ESI 中传递调用控制器的地址。

在保护模式中调用 INT 21H 更复杂些，它需要调用单独的 VMM。基本原则是首先分

配保护模式的回调(一个从保护模式过程向 VxD 过程传递控制的程序)，接着这个回调作为 INT 21H 的保护模式中断控制器被安装。最终的结果是，当保护模式应用程序执行 INT 21H 时，控制被传给了 TRAPINT。

14.3 编 程 细 节

在 Sys_Critical_Init 期间，TRAPINT 使用 EAX 中的中断号来调用 Get_PM_Int_Vector ( )。这个服务在 ECX:EDX 中返回当前控制器的地址。接着，TRAPINT 通过在 ESI 中放置 PM_Int_21H 的地址以及调用 VMM 的服务 Allocate_PM_Callback ( )来分配保护模式回调。从 VMM 返回时，EAX 包含将控制传递给 TRAPINT 的程序地址。

在将线性地址转换为段和偏移量后，TRAPINT 就调用 WMM 的服务 Set_PM_Int_Vector ( )将这个回调地址作为 INT 21H 的控制器进行安装。程序 V86_Int_21H 很简单，因为 VMM 隐藏了连接到原始的 V86 模式控制器的细节。

INT 21H 需要处理两个不同的控制器：V86_Int_21H 处理 V86 模式的控制器和 PM_Int_21H 处理保护模式的控制器。这两个程序的代码是类似的，不过 V86_Int_21H 更简单。当 PM_Int_21H 必须进行自己的工作时，VMM 隐藏了连接到原始的 V86 模式控制器的细节。

14.4 总 结

Windows 没有被设计为实时的操作系统，并且一些时间函数的行为也对它有影响。要得到精确的时间，最好使用多媒体函数。

捕获软件中断的 VxD 对以前介绍的捕获 VxD 做了相当简单的修改。虽然要捕获每个 INT 21H 比较困难，但实际上捕获软件中断却相对简单。典型的驱动程序一般只需要捕获一到两个中断。

以下程序说明 VxD 怎样在 V86 和保护模式中调用软件中断。

列表 14.1 trapint.asm

```
PAGE   58, 132
TITLE  trapint.asm
; ==============================================
       .386p
; ***********************************************
;             INCLUDE
; ***********************************************
.XLIST
       include   VMM.INC
       include   VPICD.INC
```

```
        include    DEBUG.INC
.LIST
; ************************************************
;      VIRTUAL  DEVICE  DECLARATION
; ************************************************
Declear_Virtual_Device TRAPINT, 1, 0, TRAPINT_Control,
Undefined_Device_Id,
Undefined_Init_Order,
; ************************************************
; Inittialization   data

VxD_IDATA_SEG

VxD_IDATA_ENDS

; ************************************************
; Global   data

VxD_DATA_SEG

DOS_INT      EQU   21h
Orig_PM_Int_Sel    dd    0
Orig_PM_Int_Off    dd    0

VxD_DATA_ENDS

VxD_LOCKED_DATA_SEG

VxD_LOCKED_DATA_ENDS

; ************************************************
; Initialization  code

VxD_ICODE_SEG
; ************************************************
;  TRAPINT_Sys_Critical_Init
;  DESCRIPTION:
;  ENTRY:     EBX is SYS VM handle
;  EXIT:
;  USES:
; ================================================
```

```
BeginProc  TRAPINT _Sys_Critical_Init
        mov         eax, DOS_INT
        mov         esi, OFFSET32 V86_Int_21h
        VMMcall     Hook_V86_Int_Chain
        jnc         hook_v86_okk
        Debug_Out   'Hook_V86_Int_Chain  Failed'
        jmp         SHORT  init_err

hook_v86_okk:
        mov         eax, DOS_INT
        VMMcall     Get_PM_Int_Vector
        mov         [Orig_PM_Int_Sel], ecx
        mov         [Orig_PM_Int_Off], edx

        mov         esi, OFFSET32 PM_Int_21h
        VMMcall     Allocate_PM_Call_Back
        jnc         alloc_okk
        Debug_Out   'Allocate_PM_Call_Back  Failed'
        jmp         SHORT  init_err

Alloc_ok:
        mov         ecx, eax
        movzx       edx, cx
        shr         ecx, 16
        mov         eax, DOS_INT
        VMMcall     Set_PM_Int_Vector
        clc
        ret

init_err:
        stc
        ret

EndProc TRAPINT_Sys_Critical_Init

VxD_ICODE_ENDS
; *************************************************

VxD_CODE_SEG

; *************************************************
:  TRAPINT  _Control
```

```
; DESCRIPTION:dispatch control messages to correct handlers
; ENTRY:
; EXIT:  Carry clear if no error
; USES:
; ==================================================

BeginProc  DMABUF_Control
      Control_Dispatch  Sys_Critical_Init, TRAPINT _ Sys_Critical_Init
      Clc
      Ret
EndProc  DMABUF_Control

; ************************************************
; V86_Int_21h
; DESCRIPTION:
; ENTRY: EAX=interrupt  number
       EBX=current  VM  handle
;       EBP→Client  Register  Structure
; EXIT:
; USES:
; ==================================================

BeginProc  V86_Int_21h
      cmp     [ebp, Client_AH], 41h
      je      process_v86_del

      stc
      ret

process_v86_del:
      or      [ebp, Client_Eflags], CF_Mask
      mov     [ebp, Client_AL], 05h
      clc
      ret

EndProc  V86_Int_21h


BeginProc  PM_Int_21h
      cmp     [ebp, Client_AH], 41h
      je      process_pm_del
```

```
        mov     ecx, [Orig_PM_Int_Sel]
        mov     ecx, [Orig_PM_Int_Off]
        VMMjmp  Simulate_Far_Jmp

process_pm_del:
        or      [ebp, Client_Eflags], CF_Mask
        mov     [ebp, Client_AL], 05h
        VMMcall Simulate_Iret
        ret

EndProc PM_Int_21h

VxD_CODE_ENDS

END

; End of File
```

列表 14.2　trapint.def

```
LIBRARY         TRAPINT
DESCRIPTION     'Win386 TRAPINT Device(Version1.0) '
EXETYPE         DEV386
SEGMENTS
                _LTEXT  PRELOAD  NONDISCARDABLE
                _LDATA  PRELOAD  NONDISCARDABLE
                _ITEXT  CLASS  'ICODE'  DISCARDABLE
                _IDATA  CLASS  'ICODE'  DISCARDABLE
                _TEXT   CLASS  'PCODE'  NONDISCARDABLE
                _DATA   CLASS  'PCODE'  NONDISCARDABLE
EXPORTS
                TRAPINT _DDB   @1
```

读者意见反馈卡

请您认真填写本卡并寄给我们。对于发现本书中技术问题的读者，我们另有答谢。

1. 您对本书的总体感觉:

□满意 □一般 □不满意

2. 您认为本书的层次结构:

□很好 □一般 □不好

3. 您认为本书的语言文字水平:

□很好 □一般 □不好

4. 您认为本书的版式编排:

□很好 □一般 □不好

5. 您认为本书中所涉及各项操作说明的准确性:

□准确 □较准确 □不准确

6. 您最需要哪方面的图书? ______________________

7. 您是从哪里第一次听说这本书的?

□书店 □广告 □从朋友、同事等处听说 □其他

8. 您一年中购买计算机类图书的数量:

□2~5本 □6~10本 □多于10本

9. 您使用的操作系统是:

□DOS □Windows □OS/2 □Macintosh □Unix □Linux □其他

10. 您感兴趣的计算机类新书为:

□操作系统类 □办公软件类 □程序设计语言类

□图形、图像设计类 □排版软件类 □网络技术类

□多媒体制作类 □其他

11. 您使用PC机的地方:

□家庭 □单位 □学校 □其他

12. 您是否有CD-ROM:

□有 □无

读者姓名: **单位名称:** **联系电话:**

请填好本卡后寄给:

清华大学校内金地公司(E-mail: jdsd@public.east.net.cn)

《Windows设备驱动程序技术内幕》编辑部收

邮编: 100084

联系电话: (010) 62770381 传真: (010) 62770384

如需本书可与本编辑部联系邮购(汇款请按以上地址填写，另加邮费10%)